A programozás magasiskolája

B.W. KERNIGHAN
P. J. PLAUGER

B. W. KERNIGHAN—P. J. PLAUGER

A programozás magasiskolája

MŰSZAKI KÖNYVKIADÓ, BUDAPEST, 1982

Ez a könyv B. W. Kernighan és P. J. Plauger: Software Tools
című műnek, a mű kiadására és forgalmazására vonatkozó
minden jogot birtokló Addison-Wesley Publishing Company,
Inc., Reading, Massachusetts, U.S.A., engedélye alapján
kiadott és forgalmazott magyar fordítása.

Fordította:
Bakos Tamás matematikus

Lekorálta:
Lőcs Gyula matematikus

© Műszaki Könyvkiadó, 1982
Hungarian translation

ETO: 681.326

ISBN 963 10 4218 9

Felelős szerkesztő: Oláh Gyuláné matematikus

Tartalom

ELŐSZÓ A MAGYAR KIADÁSHOZ 9

ELŐSZÓ 11

BEVEZETÉS 13

1. fejezet ELŐKÉSZÜLETEK 19
- 1.1 File-ok másolása 19
- 1.2 Karakterszámlálás 24
- 1.3 Sorszámlálás 25
- 1.4 Szavak számlálása 27
- 1.5 Tabulátorok eltávolítása 30
- 1.6 A program kézi fordítása 35
- 1.7 Néhány szó a strukturált programozásról 39
- 1.8 A RATFOR összefoglalása 39
- 1.9 A **getc** és a **putc** egyszerű változatai 42
- 1.10 Hogyan tovább? 44

2. fejezet SZŰRŐK 47
- 2.1 Tabulátorok visszaállítása 47
- 2.2 Programok paraméterezése 49
- 2.3 Átütések 51
- 2.4 Szövegtömörítés 53
- 2.5 Szövegkifejtés 59
- 2.6 Rejtjelezés 61
- 2.7 Karakterek átírása 63
- 2.8 Számok 73
- 2.9 Összefoglalás 76

3. fejezet FILE-ok 79
- 3.1 File-ok összehasonlítása 79
- 3.2 File-ok név szerinti egymáshoz rendelése 83
- 3.3 File-beillesztés 86
- 3.4 File-egyesítés (konkatenáció) 89
- 3.5 File-listázó 90
- 3.6 Többlépcsős feldolgozás: csatlakozások 94
- 3.7 File-ok dinamikus létrehozása 96

3.8 Mindezek együtt: archívum 97
3.9 További archívumkezelő parancsok 106
3.10 Programszerkezet 112
3.11 Összefoglalás 114

4. fejezet RENDEZÉS 117
4.1 Buborékos rendezés 117
4.2 Schell-rendezés 118
4.3 Szövegek rendezése 119
4.4 Gyorsrendezés 124
4.5 Nagy file-ok rendezése 128
4.6 Továbbfejlesztés 134
4.7 A funkciók szétválasztása: unique 137
4.8 Permutált index 140

5. fejezet SZÖVEGMINTÁK 147
5.1 Szövegminták 148
5.2 A megvalósítás 151
5.3 A minták előállítása 160
5.4 Néhány mérési eredmény 166
5.5 Szövegváltoztatás 167
5.6 Összefoglalás 172

6. fejezet SZERKESZTÉS 175
6.1 Amit a felhasználó lát 176
6.2 A megvalósítás 183
6.3 A vezérlőprogram 191
6.4 Puffer-reprezentáció 195
6.5 További parancsok: törlés, beszúrás, csere, sorszámnyomtatás, átvitel . 201
6.6 A helyettesítő parancs 204
6.7 Bemenet/kimenet 207
6.8 Globális parancsok 211
6.9 A főprogram 214
6.10 Munkafile-ok 219
6.11 Összefoglalás 226

7. fejezet FORMÁZÁS 229
7.1 Parancsok . 230
7.2 A program szerkezete 234
7.3 Parancselemzés 235
7.4 A lapok formátuma 238
7.5 Bekezdések . 245
7.6 Feltöltött szöveg 247
7.7 A sorvégződések kiegyenlítése 249
7.8 Középre állítás és aláhúzás 252
7.9 Mérési adatok 256
7.10 Kiterjesztések 256
7.11 Lényegesebb bővítések 259

8. fejezet MAKRÓFELDOLGOZÁS . 263
8.1 Egyszerű szöveghelyettesítés 264
8.2 Keresés a táblázatban 271
8.3 Néhány mérési eredmény 274
8.4 Paraméteres makrók . 275
8.5 A megvalósítás . 278
8.6 Feltételes makrók és más beépített műveletek 286
8.7 Alkalmazások . 290

9. fejezet A RATFOR—FORTRAN fordító 295
9.1 Szervezési kérdések . 296
9.2 Lexikális analízis . 298
9.3 Kódgenerálási szabályok 301
9.4 Az elemzés . 305
9.5 Kódgenerálás . 310
9.6 Kiviteli rutinok . 318
9.7 Bővítések . 321
9.8 Mérési eredmények . 324
9.9 Működési statisztika 325

UTÓSZÓ . 327

Függelék. ELEMI MŰVELETEK ÉS SZIMBOLIKUS KONSTANSOK 331

A RUTINNEVEK MUTATÓJA . 333

TÁRGYMUTATÓ . 347

A könyvben leírt összes program géppel olvasható formában hozzáférhető az Addison—Wesley kiadónál.*

* A programok Magyarországon a KFKI-ben és a SZÁMALK-ban találhatók meg, illetve szerezhetők be.

Előszó a magyar kiadáshoz

Örömmel adjuk olvasóink kezébe Kernighan és Plauger könyvének magyar fordítását.

Ez a könyv nem kész programozási recepteket közöl, hanem olvasmányos formában rávezeti az olvasót arra, hogy hogyan kell nagyobb programokat és programrendszereket átgondoltan megtervezni és kivitelezni. Az egész művön az az alapgondolat vonul végig, hogy amit csak lehet, használjunk fel abból, amit előttünk létrehoztak, és ne kezdjünk mindig mindent elölről. Ehhez természetesen az szükséges, hogy az, amit létrehozunk, olyan eszköz, építőkő legyen, amelyet később akár sajátmagunk, akár mások — esetleg eltérő követelményrendszer mellett is — fel tudjunk majd használni.

Külön említésre méltó értéke a könyvnek, hogy a benne közölt programok mind igen alaposan ki vannak próbálva, és sikeresen futottak különböző számítógépeken. A Számítástechnika Alkalmazási Vállalat ezeket a mű eredeti kiadójától beszerezte, ESZR gépekre történő adaptációjukat pedig Dubla László, az MTA Központi Fizikai Kutató Intézet programozó matematikusa végezte el. Néhány kisebb illesztési probléma megoldása után valamennyi program működőképesnek bizonyult.

A könyv fordítása során törekedtünk a szabványos, vagy ennek hiányában a legelterjedtebb magyar szakkifejezések használatára. A programok és egyéb programozási objektumok (pl. változók) elnevezéseit azonban nem magyarosítottuk, részint mivel ez hatalmas munkát és egyben hibaforrást is jelentett volna, részint mivel a rendelkezésünkre álló programanyag az angol elnevezéseket használja. Ez az angolul nem tudó Olvasónak némi kényelmetlenséget jelent majd, minthogy az elnevezéseknek az ő számára nincs mnemonikus értéke.

Úgy érezzük, hogy ez a könyv hasznos segédanyaga lesz az elektronikus számítógépek haladó szintű programozási tanfolyamainak.

A Kiadó

Előszó

Ez a könyv azt kívánja megtanítani, hogyan lehet jó eszközként használható jó programokat írni. Célját egy olyan teljes készlet bemutatásával igyekszik elérni, amelynek minden eleme tervezési és megvalósítási mintaként szolgál. A programok nem mesterségesek és nem öncélú játékok, hanem olyan eszközök, amelyek már hasznosnak bizonyultak más programok előállításában. A legtöbbjüket a napi munkánk során használjuk, és ezek alkotják számítógép-felhasználásunk legnagyobb részét. A programok teljesen befejezettek — nem csupán algoritmusok és vázlatok — és működőképesek is: mindegyiküket az itt közölt és egyben a gép számára is érthető szöveg alapján próbáltuk ki. A programok könnyen olvashatók: egy RATFOR-nak (racionális FORTRAN-nak) nevezett, strukturált nyelven vannak megírva, amelyet mindenki megért, aki akár FORTRAN-t, PL/l-et, COBOL-t, ALGOL-t, PASCAL-t akár egy másik hasonló nyelvet ismer. (A RATFOR-t könnyű FORTRAN-ra vagy PL/1-re fordítani: az egyik bemutatott eszköz éppen egy olyan előfeldolgozó, amely a RATFOR szöveget automatikusan FORTRAN-ra fordítja.) Ami pedig a legfontosabb, a programokat a felhasználókkal és egymással való jó együttműködés szem előtt tartásával terveztük, ezért fogadhatók el eszközökként.

A könyv pragmatikus. A felülről lefelé haladó (top-down) tervezést tervek elemzésével tanítjuk. A strukturált programozást strukturált programokon mutatjuk be, A hatékonyság és megbízhatóság kérdéseit a ténylegesen elvégzett tesztek alapján tárgyaljuk. A portabilitás problémáját úgy oldjuk meg, hogy egyrészt egy széleskörűen hozzáférhető nyelvet használunk, másrészt az elkerülhetetlen, rendszerfüggő tulajdonságokat egy kis, igen pontosan körülhatárolt rutincsomagba gyűjtjük össze, amely bármely adott működési környezetben könnyen létrehozható. Az itt bemutatott programok mindegyike legalább két gépen futott változtatás nélkül; a nagyobbak pedig egy IBM 370-en, egy Honeywell 6070-en és egy PDP-11-en.

A jó programozás alapelveit nem mint absztrakt kinyilatkoztatást, hanem mint ténylegesen működő programok környezetébe ágyazott leckéket mutatjuk be. Például nincs külön fejezet a „hatékonyság"-ról, hanem a könyvön végigvonulnak azok a hatékonysággal kapcsolatos észrevételek, amelyek egy-egy fejlesztés alatt álló konkrét programmal kapcsolatosak. Hasonlóképpen nincs „felülről lefelé haladó tervezés", „strukturált programozás" vagy „tesztelés és hibajavítás" c. fejezet sem. Ezeket az elveket szükség szerint minden bemutatott programmal kapcsolatban a gyakorlatban alkalmazzuk.

A könyv, érzésünk szerint, programtervezői vagy haladó szintű programozási tanfolyam anyagaként való felhasználásra alkalmasabb, mint a hagyományos assemblereken, fordítóprogramokon és betöltő programokon alapuló szokásos anyagok, mivel az itt szereplő programok mind méretüket, mind természetüket tekintve olyanok, amilyenekkel a programozók később a gyakorlatban találkozni fognak.

A szöveg felhasználható bármilyen programozási tanfolyam kiegészítő anyagaként is. Az egyetlen feltételezett előismeret: programozási gyakorlat egy magas szintű nyelven. Hivatásos programozók a könyvet a jó programozási technika vezérfonalának és kipróbált hasznos programok forrásának tekinthetik. Számos gyakorlat szerepel a könyvben az anyag megértésének ellenőrzésére, valamint a bemutatott elvek és programok általánosítása céljából.

Bármely területen csak akkor haladhatunk jelentős mértékben előre, ha építünk mások munkájára is. Ezzel szemben a programozást még ma is „kisipari" tevékenységként űzik, mivel a programozók ragaszkodnak ahhoz, hogy újra feltalálják ugyanazokat a programokat minden egyes új alkalmazáskor, ahelyett, hogy felhasználnák a már létezőket. Mi azt reméljük, hogy sikerül egy szemléletmódot kialakítani olyan jó programok tervezésével és írásával kapcsolatban, melyek széleskörűen használhatók; elterjeszteni a meglevő eszközök használatát és megvilágítani azt, hogy hogyan lehet egy adott környezetet továbbfejleszteni maximális hatékonysággal és minimális munkaráfordítással.

Minden programozással kapcsolatos könyv elismeréssel kell, hogy adózzon a programozási folyamat tökéletesítésén tevékenykedő úttörőknek. E. W. DIJKSTRA C. A. R. HOARE, D. E. KNUTH, D. L. PARNAS és N. WIRTH ránk gyakorolt hatását a legnagyobb mértékben elismerjük.

Hálásak vagyunk barátainknak és kollégáinknak az anyag gondos átolvasásáért, alkotó bírálataikért, valamint minden várakozást meghaladó folyamatos, lelkes támogatásukért. Közöttük DENNIS RITCHIE több kéziratrészt nézett át gondosan, javaslatai alapján mind a szöveg, mind a programok tökéletesebbek lettek.

Az általa, az UNIX-hoz tervezett C nyelv volt a RATFOR modellje; szövegszerkesztőnk is az ő szerkesztőjének struktúráját követi. Tőle származik a 8. fejezetben szereplő makrofeldolgozó prototípusának tervezése és megvalósítása is. Végül a programok kipróbálásakor kivételes tehetséggel hívta fel a figyelmet a lehetséges hibákra, és közreműködött a programok több gyenge pontjának kiküszöbölésében. Segítségét igen nagyra értékeljük.

Köszönetünket fejezzük ki STU FELDMAN-nak és DOUG MCILROY-nak a kéziratok többszöri javításáért, AL AHO-nak, BRENDA BAKER-nek, RICK BECKER-nek, PHYLLIS FOX-nak, DAN FRANKLIN-nak, TOM GIBSON-nak, KEN THOMPSON-nak, TONY WASSERMAN-nek és ED YOURDON-nak a kézirat elkészítésének különböző fázisaiban fűzött megjegyzéseiért, valamint MIKE LESK-nek és JOE OSSANA-nak a szöveg szerkesztésében nyújtott segítségért.

Végül örömünkre szolgál, hogy kifejezhetjük elismerésünket az UNIX operációs rendszerért, amelyet KEN THOMPSON és DENNIS RITCHIE fejlesztett ki a Bell Laboratóriumban. A szöveg írásakor, a programok tesztelésekor és a kézirat nyomdai szedésekor kizárólag az UNIX-ot használtuk.

A leírt eszközök jó része UNIX modelleken alapszik. Ami a legfontosabb, ötleteink és filozófiánk UNIX felhasználókként szerzett tapasztalatainkon alapulnak. Az általunk használt operációs rendszerek között az UNIX az egyetlen, amelyik valóban segítséget nyújt egy feladat elvégzéséhez, nem pedig eggyel növeli a leküzdésre váró akadályok számát. Az UNIX széles körű és egyre gyorsuló terjedése bizonyítéka annak, hogy véleményünkkel nem állunk egyedül.

BRIAN W. KERNIGHAN
P. J. PLAUGER

Bevezetés

Két kérdést tárgyalunk ebben a könyvben: hogyan írjunk jó eszközként használható programokat és ennek során hogyan programozzunk jól.

Mit értünk *eszközön?* Tegyük fel, hogy valakinek egy 5000 soros FORTRAN-programot kell áttelepítenie egyik számítógépről a másikra, és meg kell találnia az összes **format** utasítást, hogy biztos legyen benne, hogy azok az új gépen jól működnek. Hogyan kellene ehhez hozzáfogni?

Az egyik lehetőség, hogy egy programlistát piros ceruzával megjelölgetünk. Ám nem kell túl nagy képzelőerő ahhoz, hogy észrevegyük, mi a baj száz oldal sornyomtató papír piros ceruzás jelölgetésével. Értelmetlen, unalmas robotmunka ez, rengeteg hibalehetőséggel. És miután megtaláltuk az összes **format** utasítást, még mindig nem jutottunk messzire, mivel a piros jelölések a gép számára nem olvashatók.

Egy másik megoldás, hogy írunk egy egyszerű programot, amelyik megkeresi a **format** utasításokat. Ez jobb megoldás, mivel egy ilyen program gyorsabb és pontosabb a kézi munkánál. A baj csak ott van, hogy ez a program annyira speciális, hogy a szerzője csak egyszer használja, aztán félreteszi és elfelejti. Senki más nem fogja hasznosítani a program megírásába fektetett erőfeszítést, és minden újabb alkalmazáshoz újra valami nagyon hasonlót kell majd alkalmazni.

A **format** utasításoknak FORTRAN-programokból való kikeresése egy általános probléma speciális esete: jelsorozatok megtalálása egy szövegben. Aki ma **format** utasításokat keres, az holnap **read** és **write** utasításokat fog keresni, és a jövő héten egy egészen más jelsorozatot az eddigiektől valamilyen teljesen eltérő szövegben. A piros ceruzával való jelölgetés tehát soha nem ér véget. Az általános probléma megoldása az, hogy létrehozunk egy általános célú jelsorozat-keresőt, amely felkutat egy adott jelsorozatot, és kiírja mindazokat a sorokat, ahol az előfordul. Ezután bárki azt mondhatja:

find *jelsorozat*

és ezzel már végzett is a feladattal. A **find** eszköz: használja a gépet, általános feladatot, nem pedig speciális problémát old meg, és végül használata olyan egyszerű, hogy a felhasználók inkább ezt alkalmazzák majd, minthogy maguk hozzák létre saját eszközeiket.

A programozók között túl sok a „piros ceruzás". Egyesek a szó szoros értelmében „piros ceruzásak", akik kézzel hajtanak végre olyan dolgokat, amelyeket géppel kellene. Mások átvitt értelemben azok, akik a gépet olyan ügyetlenül és nehézkesen használják, hogy ezzel az erővel akár kézileg is dolgozhatnának. Ezen könyv egyik célja az, hogy megmutassa, hogyan kell *eszközöket* létrehozni — programokat, amelyek segítenek abban, hogy a problémákat piros ceruza helyett géppel oldjuk meg, mégpedig jól. Ezt a célt úgy kívánjuk elérni, hogy nem általánosságban beszélünk, hanem valódi, működő programokat írunk: programokat, amelyekről tapasztalatból

tudjuk, hogy hasznos eszközök. E könyv *minden* programját gondosan kipróbáltuk és lefuttattuk, közvetlenül a könyvnek a gép számára is olvasható szövegéből. Mindegyik program változtatás nélkül legalább két különböző gépen futott; a nagyobb programokat három gépen futtattuk: a PDP-11-en, a Honeywell 6070-en és az IBM 370-en.*

A könyv másik célja a *jó* programok írásának bemutatása. Reméljük, hogy a könyv olvasása során sikerül a következő elveket közvetítenünk: jó tervezés, hogy könnyen karbantartható és módosítható programok készüljenek; emberi szemléletmód, hogy használatuk kényelmes legyen; megbízhatóság, hogy a jó megoldást szolgáltassák; és végül hatékonyság, hogy érdemes legyen futtatni őket.

Napjainkban igen sok kifejezés terjedt el a programozási folyamat javításának leírására: strukturált programozás, felülről lefelé haladó tervezés, strukturált tervezés, személytelen programozás, és még sorolhatnánk. Például szűkebb értelemben a strukturált programozás a vezérlő utasítások korlátozott halmazával való programozást, a **goto** utasítások elkerülését jelenti. Tágabb értelemben viszont a programozás általános folyamatát és ezen belül a program vezérlési szerkezetének gondos körültekintéssel való kialakítását foglalja magában, ami egyaránt vonatkozik a program logikájára, az adatok szervezésére, valamint a külvilággal való kapcsolatra. A felülről lefelé haladó tervezés és a lépésenkénti finomítás a programozási feladatot a legáltalánosabb fogalmak segítségével közelíti meg, majd ezeket egyre konkrétabb és részletesebb tevékenységekké bővíti mindaddig, amíg a teljes program el nem készül. A strukturált tervezés egy program vagy rendszer általános tervezésének irányítása oly módon, hogy az egyes részek jól illeszkedjenek egymáshoz, de ugyanakkor megfelelő mértékben függetlenek is maradjanak, hogy külön-külön módosíthatók legyenek. A személytelen programozás azt teszi lehetővé mások számára, hogy programjainkat elolvashassák, és képesek legyenek anélkül javítani rajtuk, hogy mi ezen megsértődnénk. A fenti elvek mindegyike kézzelfoghatóan javíthatja a programozó termelékenységét és az előállított kód minőségét.

Tévedés azonban azt hinni, hogy bármelyik technika merev alkalmazása önmagától jó programokat eredményez. Nem hisszük, hogy a programozást meg lehetne tanulni úgy, hogy általános frázisokat olvasunk a jó programozásról. Ugyancsak nem elegendő, ha kis, mesterséges példákat tanulmányozunk. Ahelyett, hogy a strukturált programozáshoz és a felülről lefelé haladó tervezéshez hasonló módszereket absztrakt elvekként vezetnénk be, megkíséreltük mindegyiknek a lényegét megragadni és a programjainkban gyakorlattá tenni. Ily módon az olvasó tisztába jöhet a jelentésükkel, valódi problémákra való alkalmazásukkal és azzal, hogy milyen előnyök várhatók tőlük. Ez a könyv nem a strukturált programozásról vagy bármelyik más módszerről szól, hanem ezek felhasználását mutatja be.

Így a **goto** utasítás használatát nem nagyrabecsülésünk jeléül kerüljük, nem is annak a reményében, hogy ezzel automatikusan jó programokat állítunk elő, hanem azért, mert tapasztalatunk szerint a **goto** utasítások használata gyakran hátrányos. Azért törekszünk felülről lefelé tervezni, mert ez olyan programokhoz vezet, amelyek könnyebbben belőhetők, módosíthatók, és a módosítástól kevésbé károsodnak. Ami a személytelen programozást illeti, az olvasó könyvünk több mint 5000 programsorát tanulmányozhatja.

Megpróbáljuk azt is megmutatni, hogy *milyen utat jártunk be* a programok felépítése során, tehát nem csupán a befejezett terméket mutatjuk be, és – ami még rosszabb lenne – nem akarjuk azt a látszatot kelteni, hogy a végső eredményhez valami-

* Hazánkban ezeket a programokat az ESZR gépcsaládból az R40 gépen futtattuk le.

lyen mechanikus folyamat útján jutottunk. Minden programról leírjuk annak célját, vagyis azt, hogy miként kell azt megtervezni a könnyű felhasználás érdekében, hogy milyen megfontolások befolyásolják szerkezetét és megvalósítását, és végül néhányat a lehetséges megoldási változatok közül. Nem állítjuk, hogy ezek a lehető legjobb programok, vagy hogy a mi módszerünk az egyetlen út a tervezésükre és megírásukra. De még ha az olvasó más utat is választana végigkísérni hasznos programok egy összefüggő és jól megírt sorozatának fejlesztését, segítséget nyújthat az itt szereplő ötletek értékelésében és a jó programozói tulajdonságok kifejlesztésében.

Jó néhány eszközt mutatunk be az olvasónak. Ezek legtöbbje kezelhető méretű, olyan, amilyet egy ember várhatóan egy óra, egy nap vagy egy hét alatt megírhat. Nyilvánvalóan nincs lehetőségünk arra, hogy olyan hatalmas programokat, mint operációs rendszerek vagy fordítóprogramok bemutassunk; nagyon kevesen vannak, akiknek idejük, képzettségük vagy szükségük van arra, hogy ilyen rendszerek szerkezetébe belemélyedjenek. Ehelyett olyan programokra koncentráltunk, amelyekkel az olvasó nagy valószínűséggel kapcsolatba kerül, és amelyek segítséget nyújtanak bármelyik létező operációs rendszer vagy nyelv hatékonyabb felhasználásában. Ebből fontos tanulság vonható le: helyesen választott és jól tervezett, szerény méretű programokat lehet arra használni, hogy kényelmes és hatékony kapcsolatot teremtsenek nagyobb és kevésbé jól kivitelezett programok között.

Amikor csak lehetséges, az összetettebb programokat egyszerűbbekből építjük fel; sőt ha mód van rá, még a felépítést is *elkerüljük* oly módon, hogy önmagukban vagy kombinálva őket új alkalmazásokat keresünk a meglevő eszközök számára. Programjaink *együttműködnek;* együttes hatásuk sokkal nagyobb, mint amit egy olyan, hasonló programgyűjtemény hozhatna létre, amelynek tagjait nem lehet egykönnyen összekapcsolni. A könyv végére érve az Olvasó a szoftvereszközök olyan gyűjteményével ismerkedik meg, amely sok, a programozó gyakorlatában előforduló problémára ad megoldást. Néhány ezek közül csak vázlat, az a minimum, ami már hasznos; a hozzájuk tartozó gyakorlatok irányt mutatnak, hogy ezek értelmes módon hogyan fejleszthetők tovább.

Milyen eszközök ezek? A számítástechnika igen tág terület, és lehetetlen az összes alkalmazást megmutatni. Ehelyett olyan tevékenységre koncentráltunk, amely a programozásban központi jelentőségű: *programokra, amelyek más programok fejlesztését támogatják*. Ezek olyan programok, amelyeket rendszeresen használunk, legtöbbjüket nap mint nap; ezen könyv megírása során mi lényegében mindegyiket használtuk. Sőt éppen azért választottuk ezeket a programokat, mert az általunk használt rendszeren ezek teszik ki a számítógép-használat jelentős részét. Bár azt nem ígérhetjük, hogy választásunk majd minden igényt kielégít, de néhány program bizonyára közvetlenül is hasznos lesz az olvasó számára, bármely területen tevékenykedjék is. A nem közvetlenül hasznos programok tanulmányozása bepillantást és ötleteket adhat a speciális problémák megoldására, készülő jó minőségű eszközök tervezéséhez és felépítéséhez. A mi megoldásaink és az olvasó rendszerében rendelkezésre álló megfelelő programok összehasonlítása révén mindkettőt tökéletesíthetjük. Továbbá, az eszközök szintjén való gondolkodás elsajátítása arra ösztönzi az olvasót, hogy problémájának csak egyedi részét oldja meg saját maga, a többit a meglevő programokhoz kapcsolódás útján végezze el.

Akármilyen alkalmazásról legyen is szó, a legfontosabb eszköz egy jó programozási nyelv. Enélkül a programokat nagyon nehéz megírni és megérteni; több időt töltünk a nyelvvel való küszködéssel, mint eredményes munkával. Szerencsés esetben a rendszerünkben már létezik egy jó nyelv, a megfelelő támogatással. Ha ilyen nincs, létre kell hozni a sajátunkat. Ez nem azt jelenti, hogy hozzá kell kezdenünk egy for-

dítóprogram megírásához, hiszen ez a szakemberek dolga. Ehelyett azt kell megtanulnunk, hogyan lehet bármilyen létező nyelvet továbbfejleszteni egy vagy több előfeldolgozó segítségével.

Amikor programozásról írunk, az egyik probléma a programok nyelvének megválasztása. Nincs olyan nyelv, amelyet minden olvasó ismerne és amely minden gépen lefordítható és könnyen olvasható is lenne. Meg kell alkudnunk: mivel a FORTRAN széles körben hozzáférhető és jól támogatott, alapként ezt fogjuk használni könyvünkben. A legtöbb programozó képes legalábbis olvasni a FORTRAN-t, amely majdnem minden számítógépen használható. A nyelv megfelelően szabványosított ahhoz, hogy számítógépek széles skáláján változtatás nélkül futtatható programok írhatók legyenek benne. Gyakorlatilag a FORTRAN a számítástechnikában olyan, mint a „lingua franca", amely a legközelebb áll az univerzális nyelvhez.

Ám a puszta FORTRAN önmagában meglehetősen szegényes nyelv mind a programozás, mind a programok leírása szempontjából. Ezért programjainkat a FORTRAN egy egyszerű kibővített változatában írtuk, amelyet „RATFOR"-nak (racionális FORTRAN-nak) neveznek. A RATFOR-ban korszerű vezérlési utasítások vannak, hasonlóan a PL/1-hez, a COBOL-hoz, az ALGOL-hoz vagy a PASCAL-hoz, így benne a strukturált programozás kényelmesen valósítható meg. RATFOR-ban könnyű programot írni, olvasni és azt megérteni, továbbá a programok egyszerűen fordíthatók FORTRAN-ra, PL/1-re vagy hasonló, magas szintű nyelvekre. Eltekintve néhány új utasítástól, mint az **if-else, while** és **repeat-until,** a RATFOR tulajdonképpen FORTRAN. Ha valaki csak nagy vonalakban is ismeri a FORTRAN-t, könnyen megszokja a RATFOR-t is.

Ha az olvasó más nyelvekhez szokott, nem okoz számára nehézséget, hogy programjainkat kövesse, mivel a jól strukturált programok a legtöbb nyelven hasonlóan festenek. Elkerültük a FORTRAN túlzottan egyéni sajátosságainak nagy részét, az elkerülhetetleneket pedig jól körülhatárolt modulokba csomagoltuk.

Az első fejezet bemutatja a RATFOR-nyelv legnagyobb részét, példák és tömör összefoglalás kíséretében. A 9. fejezetben felépítünk egy RATFOR—FORTRAN fordítót. Bemutatunk olyan eszközöket is, amelyek tökéletesíthetik ezt, mint pl. a 8. fejezetben tárgyalt makrófeldolgozó vagy a 3.-ban ismertetett file-beillesztő program.

Az előfeldolgozó használata a FORTRAN-hoz kényelmes megoldás. Ezáltal megőrizzük a FORTRAN előnyeit — a nyelv univerzális, portábilis és viszonylag hatékony —, ugyanakkor elrejtjük legkomolyabb hátrányait, vagyis úgy kezeljük a FORTRAN-t, mintha egy assembly nyelv lenne, amelyik bármely gépen rendelkezésre áll. Ha olyan környezetben dolgozunk, amely elsősorban FORTRAN nyelvre alapozott, ez a megközelítés igen hatékony lehet. De ha még rendelkezésre is áll egy, a RATFOR-nál jobb nyelv, a programokat azzal a biztonságérzettel adaptálhatjuk, hogy a kód ellenőrzött, és azt hajtja végre, amit elvárunk tőle.

Meglepően nagy azoknak a programoknak száma, amelyeknek egy bemenő és egy kimenő adathalmazuk van, és valamilyen hasznos transzformációt végeznek az átmenő adatokon. Ezeket a programokat *szűrőknek* nevezzük. Némelyik szűrő annyira egyszerű, hogy alig tekintenénk őket eszközöknek, mégis, együtt dolgozó szűrők egy gondosan kiválasztott készlete meglehetősen bonyolult feldolgozásokhoz segíthet hozzá. Több kisebb szűrőt gyűjtöttünk össze a 2. fejezetben, köztük egy hatékony jelátkódoló programot.

Nem minden program szűrő. A 3. fejezet olyan programokkal foglalkozik, amelyek bonyolultabb kapcsolatban vannak környezetükkel. Ilyen pl. a fent említett file-beillesztő program vagy egy file-összehasonlító, egy több file-ból álló listázó és egy archívumrendszer file-ok halmazainak kezelésére. Programoknak egyik környe-

zetből a másikba való átvitelének fő problémája pontosan az, hogy azok hogyan kommunikálnak a helybeli operációs rendszerrel. Ezt a problémát úgy hidalhatjuk át, hogy meghatározzuk a környezettel való kapcsolattartás elemi műveleteinek egy szűk halmazát. Összes programunkat ezen alapműveletek felhasználásával írtuk, így az operációs rendszertől való függőség néhány szubrutinra korlátozódik. Az ezeket használó programokat bármilyen rendszerbe át tudjuk vinni, ahol ezek az elemi műveletek megvalósíthatók.

Néhány szűrő elég nagy ahhoz, hogy külön fejezetet igényeljen. Ebbe a kategóriába tartozik a 4. fejezet rendezőprogramja, a jelsorozat-kereső és -helyettesítő programok az 5. fejezetben és a 9. fejezet RATFOR—FORTRAN fordítóprogramja. A jelsorozat-kereső felhasználja a 2. fejezetben található jelátkódoló program nagy részét, hogy karakterosztályokat ismerjen fel, amelyek a számára megadható jelsorozatoknak csupán egy részét képezik. (Megjegyezzük, hogy a jelsorozat-kereső sokkal többre képes, mint a **format** utasítások megtalálása.) Bár ezek a szűrők a programfejlesztés irányában kiélezettek, a szűrési elv bármilyen alkalmazásban felhasználható. Azt a szemléletmódot erősíti, hogy egy program csupán részlete egy nagyobb folyamatnak, és hogy e részleteknek egyszerűeknek és könnyen összekapcsolhatóaknak kell lenniük. Tudatosítja azt az elvet is, hogy a file-oknak és a be-, ill. kiviteli eszközöknek lényegében tetszőleges módon párosíthatóknak kell lenniük, és így bármelyik program ezek akármelyikét használhatja.

A 6. fejezet egy szövegszerkesztő programot tartalmaz, amely sokoldalúbb az időosztásos rendszerekben általában találhatóknál. A szerkesztő tartalmazza az 5. fejezet jelsorozat-keresőjének nagy részét, így ugyanazokat a jelsorozat-osztályokat ismeri fel, mint amaz. Ha más, bemutatott programokban használjuk, olyan feladatok megoldására képes, amelyek egyébként egy külön program írását igényelnék. A szerkesztőprogram hasznosnak bizonyul még nem interaktív környezetben is.

A 7. fejezet egy szövegformázó programot tartalmaz, amelyik egy (lényegesen szűkebb) változata annak a programnak, amelyet ezen könyv nyomtatására használtunk. Amint már említettük, a 8. fejezetben egy szerény, de hasznos makrófeldolgozó található, amely felhasználható bármilyen programozási nyelv bővítésére. Végül a 9. fejezet a RATFOR előfeldolgozót tartalmazza.

Ebből az összefoglalóból úgy tűnhet, hogy túl nagy súlyt helyezünk a szövegkezelésre. Ám a számítástechnika nem csupán „számdarálásból, fordítókból, assemblerekből és betöltőkből" áll, amelyeket annyira fontosnak tartanak sok haladó programozói tanfolyamon. A programozók mindennapi munkájának nagy része szövegfeldolgozás — forrásprogramok szerkesztése, bemenő adatok előkészítése, eredmények vizsgálata, dokumentációírás. Ezek a tevékenységek a programozás középpontjában állnak; amennyire lehet, gépesíteni kell őket. A programfejlesztés az a terület, ahol az eszközöknek a legnagyobb szerepük lehet. És mivel a szövegfeldolgozó programok igen széles mérethatárok között változhatnak, a programozási technikáknak legalább olyan széles spektrumát képviselik, mint a nyelvfeldolgozók vagy a numerikus programok.

Látni fogjuk, hogy a könyv szerkezete az alkalmazásokhoz és nem a programozás különböző fázisaihoz igazodik. Ez a mű nem algoritmusokkal, adatszerkezetekkel vagy valamelyik programozási nyelvvel foglalkozó kézikönyv. Nem találunk benne külön fejezeteket olyan népszerű témákról, mint a tervezés, kódolás, tesztelés, hibakeresés, hatékonyság, emberi szemléletmód, dokumentáció stb. Eszközök létrehozásával, méghozzá színvonalas létrehozásával foglalkozik. A programozási munka említett oldalai különböző mértékben minden programmal kapcsolatban felmerülnek, és csakis úgy tarthatjuk őket szem előtt, hogy beszélünk róluk a programok

írása során. A továbbiakban megpróbáljuk átadni az olvasónak az eszközkészítés általunk helyesnek tartott megközelítését, hogy a későbbiekben képessé váljon saját eszközeinek tervezésére, elkészítésére és használatára.

BIBLIOGRÁFIAI MEGJEGYZÉSEK

A jó programozás egyik legbefolyásosabb terjesztője, akinek a nevét a leggyakrabban kötik össze a strukturált programozással: E. W. DIJKSTRA. Célszerű elolvasni az O. J. DAHL—E. W. DIJKSTRA—C. A. R. HOARE *Structured Programming* (Academic Press, 1972) című könyvet*. A *The structure of the* THE *multiprogramming system* [Communications of the ACM (CACM), 1968. május] című cikkében DIJKSTRA a kapcsolattartás elemi műveleteinek egy hierarchikus operációs rendszer konstrukciójában való szerepét tárgyalja.

A PASCAL számottevő hatással volt a számítástechnikai gyakorlatra; a nyelv különösen alkalmas strukturált programozásra és adatszerkezetek vezérlésére. Olvassuk el a *Systematic Programming: An Introduction* (Prentice-Hall, 1973) című könyvet, amelyet N. WIRTH, a PASCAL tervezője írt. A *Computing Surveys* programozásról szóló különkiadása (1974. dec.) szintén több olyan cikket közöl — köztük egyet WIRTH-től —, amelyet érdemes elolvasni**.

A programozás folyamatáról és nagy rendszerek fejlesztésének problémáiról szóló kiváló esszék találhatók F. P. BROOKS *The Mythical Man-Month* (Addison—Wesley, 1974) című könyvében. A „személytelen programozás" kifejezést G. M. WEINBERG vezette be a *The Psychology of Computer Programming* (Van Nostrand Reinhold, 1971) című lebilincselő könyvében.

A strukturált tervezés úttörője L. L. CONSTANTINE. Az alapvető elvek összefoglalását W. P. STEVENS—G. J. MYERS—L. L. CONSTANTINE *Structured design* (IBM Systems Journal, 1974. április) című cikkében találhatjuk meg. MYERS könyve a *Reliable Software Through Composite Design* (Petrocelli/Charter, 1974) a témát részletesebben tárgyalja. Egy későbbi mű a *Structured Design* (Yourdon, 1975), amelyet E. YOURDON és L. L. CONSTANTINE írt.

A B. W. KERNIGHAN-től és P. J. PLAUGER-től származó *The Elements of Programming Style* (McGraw-Hill, 1974) című könyv részletesen tárgyalja a számítógépes programok tökéletesítését, és számos példát hoz publikált FORTRAN és PL/1 programokból.

* O. J. DAHL—E. W. DIJKSTRA—C. A. R. HOARE: Strukturált programozás (Műszaki Könyvkaidó, Budapest, 1978).

** Ajánljuk még a következő könyveket: G. M. WEINBERG: A számítógép-programozás pszichológiája (Közgazdasági és Jogi Könyvkiadó, 1979), K. JENSEN—N. WIRTH: A PASCAL programozási nyelv (Műszaki Könyvkiadó, Budapest, 1982) *(A kiadó)*.

1. fejezet

Előkészületek

Ez a fejezet kissé formalizált bevezetést tartalmaz a RATFOR programozási nyelvbe, továbbá összefoglalja a könyvben később használt elveket és konvenciókat. Bemutat ezen kívül néhány kicsiny, de hasznos programot, amelyek a tárgyalást kézzelfoghatóvá teszik. Nem tudunk teljes programokat bemutatni anélkül, hogy néha egyes elveket megmagyarázásuk előtt ne használnánk, ezért induláskor az olvasónak „becsületszóra" el kell hinnie néhány dolgot, különben belebonyolódnánk magyarázataink magyarázásába. Bízzék bennünk!

1.1 FILE-OK MÁSOLÁSA

Elsőként azt a problémát érintjük, hogy hogyan kommunikál egy program a környezetével. Mivel sok programunk szövegkezeléssel kapcsolatos, az egyik alapművelet jelsorozatok beolvasása valamilyen forrásból. Ennek érdekében bevezetjük a **getc** függvényt, amelyik beolvassa a *következő karaktert,* amelyet azután a függvény értékeként visszaad; valahányszor behívjuk, új karaktert kapunk. Most nem törődünk azzal, hogy a jelek honnan jönnek, bár feltehetjük, hogy kártyaolvasóról, interaktív terminálról vagy valamilyen háttértárról (pl. mágneslemez) származnak. Nem beszélünk arról, hogy milyen jelkészletre gondolunk. Nem törődünk a hatékonyság kérdéseivel sem, bár tisztában vagyunk azzal, hogy a karakterenként való olvasás költségesnek tűnik. Egyelőre annyi részletet, amennyit csak lehet, a szőnyeg alá söprünk.

Következőként bevezetjük a **putc** függvényt, a **getc** inverzét. A **putc** egyetlen karaktert visz át valahova, pl. sornyomtatóra, terminálra vagy lemezre. Most is túltesszük magunkat a pontos részleteken és a művelet hatékonyságán. A fő szempont az, hogy a **getc** és a **putc** együttműködik — a **getc** által beolvasott karaktereket ki lehet vinni valahova a **putc** segítségével.

Ha ez a két alapművelet rendelkezésünkre áll, meglepő mennyiségű hasznos munkát tudunk végezni anélkül, hogy többet tudnánk megvalósításukról. Egyik legegyszerűbb példa, ha a **getc—putc** pár egy ciklus belsejében szerepel:

```
while (getc(c) nem az input vége)
        call putc(c)
stop
```

Ez a program átmásolja bemenetét a kimenetre* és megáll. Egyszerű feladat, amelyet megfelelően egyszerű program hajt végre. Való igaz, hogy végső soron valakinek majd foglalkoznia kell a jelkészlet kiválasztásával, a bemenet végének felismerésével,

* A továbbiakban az input és az output helyett — ahol csak azt a szövegkörnyezet engedi — a bemenet, ill. a kimenet kifejezést használjuk *(A fordító).*

a hatékonysággal stb., de a legtöbb programozónak ez nem feladata, mert a **getc** és **putc** elrejtik a részleteket. (Ha azt akarjuk tudni hogyan, a fejezet végén megtalálhatjuk a **getc** és **putc** egyszerű változatait.)

A **getc**-hez és **putc**-hoz hasonló függvényeket elemi műveleteknek nevezzük — olyan függvények ezek, amelyek a külvilággal tartják a kapcsolatot. Ezek hívják be a későbbiekben az input és output rutinokat, bármilyenek legyenek is azok az adott operációs rendszerben. Azon program számára, amely ezeket használja, a **getc** és **putc** határozza meg a karakterek szabványos belső alakját, és általános be- és kiviteli mechanizmust ad meg, amely ugyanolyanná tehető számos különböző számítógépen. Ha ilyen elemi műveleteket használunk, tervezhetünk és írhatunk programokat anélkül, hogy ezek különösebben függnének bármelyik operációs rendszer egyéni sajátosságaitól. Az elemi műveletek elszigetelik a programot az operációs rendszer által képviselt környezetétől, és biztosítják, hogy kis, jól definiált halmazt alkotva az a magasabb szintű feladat, amelyet végre kell hajtani, világosan kifejezhető legyen velük.

Az előbb bemutatott programot pszeudokódban írtuk. Ez gépi nyelvre emlékeztet, de a túlzott részletességet elkerüli oly módon, hogy időnként köznyelvet használ. Ez lehetővé teszi számunkra, hogy a program egy részét megírjuk, mielőtt minden részletét kidolgoznánk. Nagyobb programok esetében érdemes pszeudokóddal kezdeni, majd lépésenként finomítani azt mindaddig, amíg a gép számára teljesen végrehajthatóvá nem válik. Ilyen módon áttekinthetjük és magas szinten javíthatjuk a tervezést anélkül, hogy végrehajtható kódot írnánk, ugyanakkor közel maradunk egy végrehajthatóvá tehető formához. A következő lépésben megírjuk **copy**-t pontos RATFOR-ban, amely már készen áll a fordításra és a futtatásra.

```
# copy — karaktermásolás
    integer getc
    integer c
    while (getc(c) ¬=EOF)
        call putc(c)
    stop
    end
```

Magyarázat: az első sor egy megjegyzés (komment), amely megnevezi a programot, és röviden megmondja, hogy mi a feladata; ezt a fajta megjegyzést használjuk a könyv minden végrehajtható programjában. RATFOR-ban a # (számjel) bárhol is jelenik meg a sorban, egy megjegyzés kezdetét jelzi; a sor hátralevő része a megjegyzés. Ez a konvenció rugalmasabb, mint a FORTRAN-ban „C az első oszlopban”, mivel a megjegyzések és a programszöveg ugyanabban a sorban állhatnak, és könnyebb leírni, mint a PL/1-ben szereplő /*...*/ jelet, bár nem annyira általános. (A RATFOR utasítások és konvenciók rövid összefoglalását a fejezet vége felé találjuk.) Az

```
integer getc
integer c
```

sorok közönséges FORTRAN utasítások, amelyek **c**-t és **getc**-t **integer** típusúnak deklarálják. Programjainkban minden változót és függvényt deklarálni fogunk, mert ezt jó programozási gyakorlatnak tartjuk — ez bizonyos fegyelmet követel, mivel

megnehezíti új változók önkényes bevezetését, és lehetővé teszi a deklarációk mechanikus ellenőrzését. (Gondosan szétválasztjuk a függvénydeklarációkat a közönséges változók deklarációjától, hogy elősegítsük megkülönböztetésüket.) Korábban azt mondtuk, hogy **getc** egy függvény, amely egyetlen karaktert ad vissza, **c** pedig szintén egyetlen karaktert tartalmaz, de mivel kevés FORTRAN megvalósításnak van explicit karaktertípusú adata, az egészeket használjuk erre a célra. A PL/1 megengedi a karaktereket, de amint rövidesen látni fogjuk, ezek sem teljesen megfelelők.

A **copy** munkája a

```
while (getc(c) ¬= EOF)
    call putc(c)
```

sorokban megy végbe. A **while** RATFOR ciklusutasítás: ameddig a zárójelekben levő feltétel igaz, a ciklus törzse (ebben az esetben a **call putc(c)**) ismételten végrehajtódik. Amikor a feltétel hamissá válik, a ciklus véget ér, és így jutunk a **stop** utasításhoz. A **while** ciklusban szereplő és ismételten vizsgált feltétel:

```
getc(c) ¬= EOF
```

A ¬= jel a „nem egyenlő" rövidítése a RATFOR-ban is, akárcsak a PL/1-ben. A FORTRAN felhasználók számára ismerősebb a **.ne.** jelölés, bár ez kevésbé tömör. A ciklus tehát addig folytatódik, ameddig a **getc** az **EOF** jelet nem adja vissza.

A **getc(c)** a következő karaktert mint saját függvényértékét és mint **c** argumentumának értékét adja, így az értéket egyetlen utasításban vizsgálhatjuk és tárolhatjuk további felhasználás céljából. Ez nem a szokásos FORTRAN-beli felhasználás, de teljesen hibátlan. Nagyon kényelmes, ezért gyakran használjuk.

Az **EOF** *szimbolikus konstans,* amely egész értéket jelöl, amellyel kapcsolatban a **getc** és a **getc** felhasználói úgy egyeztek meg, hogy a bemenet végét jelzi, vagyis azt, hogy nincs több karakter. Nem mondjuk meg, hogy ennek mi az értéke, mivel a konkrét érték nem fontos, gépenként más és más lehet. Az egyetlen kikötés az, hogy az **EOF**-ot meg kell tudni különböztetni minden olyan valódi karaktertől, amelyet a **getc** adhat. A szimbolikus konstansok neveiben következetesen nagybetűket használunk, hogy könnyű legyen őket észrevenni; a változókat és függvényeket viszont kisbetűkkel írjuk.

Programjainkban meglehetősen sok szimbolikus konstans fordul elő; ezek nagymértékben hozzájárulnak a kód olvashatóságához. Egy pillantással felfogható, hogy a következő vizsgálat:

```
getc(c) ¬= EOF
```

mit jelent, mivel az **EOF** kifejezőbb, mint ha valamilyen konkrét szám, pl. −1 lenne. A 8. fejezet tartalmaz egy programot, amely lehetővé teszi a szimbolikus konstansok definiálását és lefordítását. Így ezeket szabadon használhatjuk a programokban; és az őket helyettesítő tényleges jelsorozatok fordítás előtt automatikusan elhelyeződnek majd a forrásprogramban. Ehhez hasonló programot használunk a könyvben található programok előfeldolgozásához.

Egy RATFOR-program tényleges futtatásához a forrásszöveget egy *előfeldolgozónak* nevezett programon kell átbocsátani, amely lefordítja a **while**-hoz hasonló vezérlő utasításokat **if** és **goto** utasításokra, a szimbolikus konstansokat definíciójukkal helyettesíti, és az olyan rövidítéseket, mint ¬=, **.ne.**-ra konvertálja. Minden

másról feltételezhetjük, hogy közönséges FORTRAN; ezt változtatás nélkül kell átmásolni, eltekintve a kimeneti sorban való megfelelő elhelyezéstől. A kapott FORTRAN utasításokat azután a szokásos módon fordítjuk le. Így a programok feldolgozási sorrendje a következő:

RATFOR-*forrás*
→ RATFOR-*előfeldolgozó*
→ FORTRAN-*fordító*
→ *futtatható program*

Ebbe a sorozatba további előfeldolgozók is beiktathatók. Egy RATFOR-előfeldolgozót a 9. fejezetben építünk fel.

A RATFOR előnye, hogy nem engedi meg, hogy a FORTRAN szintaktikus sajátosságai elfedjék a dolgok lényegét, ugyanakkor biztosítja, hogy a programot FORTRAN-ra könnyen, és egy kicsit több ráfordítással pedig bármilyen más hasonló nyelvre le lehessen fordítani. Gyakorlati szempontból hasznos a programot első alkalommal egy könnyen olvasható, magas szintű nyelven megírni, majd ezután fordítani bármilyen, a valóságban használatos nyelvre, amellyel éppen dolgozunk — legyen az FORTRAN, PL/1 vagy akár assembly nyelv. A könyvben szereplő programok csupán néhány vezérlési szerkezetet használnak, mint pl. a **while,** és csak néhány adattípust, mint pl. az **integer,** így nem nehéz mechanikusan fordítani őket az kiválasztott célnyelvre.

Például a **copy** egy FORTRAN változata a következő:

```
c copy — karaktermásolás
      integer getc
      integer c
10    if (getc(c) .eq. EOF) goto 20
          call putc(c)
          goto 10
20    continue
      stop
      end
```

A RATFOR **while** utasítása olyan ciklus, amelynél a vizsgálat a ciklusmag előtt helyezkedik el, ezt ki lehet fejezni egy, a vizsgálatra vonatkozó **if**-fel és a ciklus végén levő **goto**-val. A **while** hatáskörébe tartozó utasításokat azonos kezdőpozícióba írjuk, hogy könnyebben felismerhetők legyenek. Az **EOF** szimbolikus konstansot szintén meghagytuk, amíg el nem tudjuk dönteni, hogy mit írjunk a helyére. Ilyen helyettesítéseket addig halasztunk, ameddig csak lehet, hogy ne zavarjuk meg az olvasót ismeretlen jelentésű számokkal.

Hasonlóképpen egyszerű a **copy** a PL/1-ben is:

```
/* copy — karaktermásolás */
    copy: procedure options (main);
    declare getc entry (fixed binary) returns (fixed binary);
    declare putc entry (fixed binary);
    declare c fixed binary;
```

```
do while (getc(c) ¬= EOF);
    call putc(c);
    end;
end copy;
```

A PL/1 **do while** utasítása nagyon közel áll a RATFOR **while**-jához, így a fordítás még egyszerűbb.

Miért hasznos a **copy**? A legtöbb operációs rendszer lehetővé teszi annak megadását, hogy milyen file-ok, adathalmazok vagy fizikai eszközök felelnek meg a programban szereplő logikai egység kódszámainak vagy belső file-neveknek. Ez a megfeleltetés csak a fordítás után, a tényleges futás idején jön létre. Ez azt jelenti, hogy kéznél lehetnek futásra kész programok, amelyekről csak az utolsó pillanatban kell eldöntenünk, hogy milyen file-okat vagy eszközöket használjanak. Továbbá azt is jelenti, hogy ilyen programokat „fekete dobozokként" kezelhetünk, és belsejükről nagyrészt elfelejtkezhetünk. Ha a **getc** elemi művelet egy „standard input"-ról olvas — mint amilyen a FORTRAN 5 logikai egysége vagy a *sysin* PL/1 file — és a **putc** egy „standard output"-ra ír — mint a FORTRAN 6 logikai egysége vagy a *sysprint* PL/1 file — a megfelelő file-okhoz vagy eszközökhöz való kapcsolást futási időben végezhetjük el.

Megjegyezzük, hogy a file szónak különböző számítógép-rendszereken különböző jelentése van. Jelenleg mi általános értelemben használjuk annak a helynek a jelölésére, ahonnan információ érkezik (pl. a **getc** útján) vagy ahova információ küldhető (pl. a **putc**-vel). Ez lehet egy lemez, amelyet állandó „file-rendszer"-ré szerveztünk, vagy bármelyik be-, ill. kimeneti eszköz.

A programunk jelsorozatot másol, bármilyen forrásból, tetszőleges rendeltetési helyre. Egy, a fentihez hasonló környezetben használhatjuk arra, hogy kártyákat vigyen fel egy lemezfile-ra, listázzon egy file-t a sornyomtatón, másolatot készítsen egy file-ról — azaz egy egész sereg „rendszerháztartási" tevékenység elvégzésére.

Bár ezen munkák elvégzésére más lehetőségek is vannak, a fenti módszer általános, és lehet rá építeni. Ha szükséges, tetszőleges számú apró módosítással tökéletesíthetjük, finomíthatjuk vagy gyorsabbá tehetjük. A **copy** alapvető *eszköz*. Önmagában is hasznos, de felhasználható más, összetettebb programok építőköveként is. Ha tartjuk magunkat ahhoz a tervezési elvhez, hogy a részletek eldöntését addig halasztjuk, ameddig csak lehetséges, azáltal, hogy olyan alapvető elemi műveletek segítségével programozunk, amelyek tetszőleges forrásból olvasnak, és tetszőleges rendeltetési helyre írnak, új eszközeink illeszkedni fognak a meglevőkhöz; olyan készletet építünk fel, amelynek elemei együttműködnek.

Gyakorlatok

1.1 Explicit módon kellett deklarálnunk **getc**-t és **c**-t egésznek, különben a FORTRAN típusszabály alapfeltételezése valósnak tekintette volna őket. Miért előnyösebbek az egészek a valósaknál? Elkerülhettük volna a deklarációkat, ha a függvényt **igetc**-nek, a változót pedig **ic**-nek hívnánk. Melyik elnevezés a jobb? Előnyben részesítené-e a **nextc** elnevezést? Ha el kellene olvasnia valaki más programját, szeretné-e, hogy minden változó és függvény deklarálva legyen, vagy csak azok, amelyeket tényleg deklarálni kell? □

1.2 Miért nem deklaráljuk adatainkat **character** típusúnak a PL/1 változatban? Mi lehetne az **EOF** célszerű értéke a program tényleges futtatásakor? Javasoljunk rossz értékeket is! □

1.2 KARAKTERSZÁMLÁLÁS

Vannak esetek, amikor csupán azt akarjuk tudni egy file-ról, hogy hány karaktert, hány sort vagy hány szót tartalmaz. Ha a file állandó tárolón helyezkedik el, pl. lemezen, lehet olyan szerencsénk, hogy az operációs rendszer választ ad legalább néhányra az előbbi kérdések közül. Ha nincs ilyen szerencsénk vagy ha a „file" például egy kártyacsomag, akkor a legegyszerűbb átbocsátani egy olyan programon, amelyik megszámolja azt, amit tudni akarunk.

Ha nem tudnánk mire vélni, hogy miért akarna valaki csupán karaktereket számlálni valahol, ne aggódjunk. Ez a könyv — mint említettük — eszközkészítésről szól, és az eszközök legjobban más eszközökkel kombinálva dolgoznak. Hamarosan találunk majd alkalmazási területeket.

A karakterek számlálása a legalapvetőbb művelet:

```
# charcount — karakterszámlálás a standard inputon
    character getc
    character c
    integer nc

    nc = 0
    while (getc(c) ¬= EOF)
        nc = nc + 1
    call putdec(nc, 1)
    call putc(NEWLINE)
    stop
    end
```

Ebben a programban **nc** valóban egy egész, kizárólag számlálásra használjuk. Deklarációjának megfelelően elkülönül **c**-től és **getc**-től, amelyek karakterek. Ez a program bevezeti a karakterek deklarációját is. Bár ez néhány esetben az **integer** szinonimája lesz, mi mostantól kezdve megkülönböztetjük programjainkban a változók két típusát, és így azonnal meg tudjuk mondani, hogy egy adott változót mire használunk. Ugyanaz az előfeldolgozó, amely az **EOF**-et minden előforduláskor egy numerikus konstanssá változtatja, felhasználható arra is, hogy a **character** minden előfordulását **integer**-re változtassa a program fordítása előtt.

Az eredményül kapott szám kinyomtatására **charcount** a **putdec** eljárást hívja, amelyik egy számot nyomtatásra alkalmas jelsorozattá alakít és kiviszi a **putc** segítségével. Ily módon nem kell tudnia, hogy a kivitel ténylegesen hogyan megy végbe. (A **putdec** eljárást a 2. fejezetben mutatjuk be.) Hívásában a második paraméter a minimális mezőszélesség: a szám egy legalább ilyen hosszú mezőben, jobbra igazítva foglal helyet.

A **putdec** eljárás sohasem kényszeríti a kimeneti eszközt — akármi is legyen az — az aktuális sor befejezésére, mivel lehet, hogy több számot akarunk egy sorban elhelyezni a **putdec** többszöri hívásával. Ezért az új sort explicit módon kell kérnünk a

```
call putc(NEWLINE)
```

segítségével.

A **NEWLINE** egy létező karakter az olyan nagy kódrendszerekben, mint az ASCII vagy EBCDIC; amikor egy írógéphez hasonló eszközre küldjük, általában kocsi-vissza-soremelést eredményez. A **NEWLINE**-t szabványkarakterként használjuk a sor végének jelzésére, tekintet nélkül az információ forrására vagy rendeltetési helyére, így programjaink azonos módszerrel különböztetik meg a sorokat.

Figyeljük meg, hogy ha a bemeneti file egyetlen karaktert sem tartalmaz, akkor a **while** ciklus vizsgálata már a **getc** első hívásakor „hamis" eredményt ad, emiatt azonnal kilépés következik, és **charcount**-ban nullát kapunk, ami a helyes válasz. Ez fontos észrevétel! A **while** utasítás egyik nagy előnye, hogy a ciklustörzs előtt végez ellenőrzést, vagyis mielőtt a törzset végrehajtatná. Ily módon, ha nincs tennivaló, semmi nem történik, még ha ez azt is jelenti, hogy egyáltalán nem hajtjuk végre a ciklus törzsét. Sok — különösen FORTRAN-ban írott — program ésszerűtlenül viselkedik, amikor pl. üres inputot kap. Mi úgy írjuk programjainkat, hogy ésszerűen viselkedjenek a szélsőséges esetekben is, aminek megvalósítására egyik alapvető eszközünk a **while**.

1.3 SORSZÁMLÁLÁS

Tegyük föl, hogy a karakterek számlálása helyett bemenő jelsorozatban előforduló *sorokat* akarunk megszámolni. Miről ismerünk fel egy sort? Különösen kényelmes megoldás, hogy a **getc** felismeri a bemeneti sorok végét és **NEWLINE**-t ad vissza. (Ha a jelkészletünk nem tartalmaz **NEWLINE**-t, akkor az **EOF**-hez hasonlóan valamilyen egyezményes értéket kell használnunk, amely minden más karaktertől különbözik.) Hasonlóan, a **putc** el kell hogy fogadja a **NEWLINE**-t mint olyan jelet, amelynek hatására ki kell vinnie a szöveg egy teljes sorát.

Ily módon *bármilyen* eszközt egy „írógép"-hez tehetünk hasonlóvá, amelyen a szöveg minden sora **NEWLINE**-nal végződik. A **getc** és **putc** megteszi mindazt, ami az eszközök közötti különbségek elfedéséhez szükséges. A programoknak semmit sem kell tudniuk a különböző eszközökhöz tartozó rekordhosszakról (sőt, még az egyes eszközökön megengedett változó hosszúságú rekordokról sem); ezek mindig egyetlen karaktert dolgoznak fel mindaddig, amíg egy **NEWLINE** elő nem fordul. A **NEWLINE**-nak karakterként való kezelése (még ha az mint karakter esetleg nem is létezik egy adott jelkészletben) sok programot egyszerűbbé tesz; ezt a fogást gyakran fogjuk alkalmazni.

Ha **getc NEWLINE**-t ad vissza minden sor végén, a **linecount** programozása egyszerű:

```
# linecount — sorszámlálás a standard inputon
    character getc
    character c
    integer nl

    nl = 0
    while (getc(c) ¬= EOF)
        if (c == NEWLINE)
            nl = nl + 1
    call putdec(nl, 1)
    call putc(NEWLINE)
    stop
    end
```

Ez alkalommal a **while** törzse egy kicsit nagyobb, egy **if**-ből áll, amely az **nl=nl+1** értékadó utasítást vezérli. A bekezdések mutatják, hogy melyik programsort melyik vezérli, és szerényen, de világosan hívják fel a figyelmet a program logikai szerkezetére.

A == (kettős egyenlőségjel) a RATFOR-ban az egyenlőség szempontjából történő összehasonlításra használt szimbólum; az értékadó operátortól, az egyetlen = jeltől való megkülönböztetés céljából vezetjük be. A == jelet FORTRAN-ban **.eq.**-ra, PL/1-ben =-re fordítjuk. Mint a bevezetésben említettük, a RATFOR sok tekintetben FORTRAN. Ha nem tetszenek a ¬=-hez és a ==-hez hasonló jelölésmódok, akkor használjuk a **.ne.** és **.eq.** jelöléseket, ezek ugyanúgy elfogadhatók.

Az az elv, hogy a szöveges információ csupán karakterek sorozata, amelyben a tetszőlegesen hosszú sorokat explicit újsor-karakterek választják el, meglehetősen nyilvánvaló, hogy ha arra gondolunk, hogy hogyan dolgozik egy írógép vagy egy terminál. Ennek ellenére még mindig szokatlan sok számítógép-rendszerben, ahol a szöveget gyakran kártyákra emlékeztető, rögzített hosszúságú darabokra tördelik.

Ha rendszerünk nagyon erőlteti a rögzített hosszúságú szövegformákat, meggondolhatjuk, hogyan kell **getc**-nek és **putc**-nek működnie. FORTRAN-ban, amely alapvetően kártya vagy rekord jellegű környezetet feltételez, egy sort tekintünk egy rekordnak. A **getc**, valahányszor csak szükség van rá, beolvas egy rekordot, ebből egyenként adogatja ki a karaktereket, és minden rekord végén egy **NEWLINE**-t ad. A **putc** egy pufferben gyűjti a karaktereket, amíg meg nem hívjuk egy **NEWLINE**-nal, amikor is kiírja a teljes rekordot (a **NEWLINE**-tól eltekintve). A puffereknek olyan nagyoknak kell lenniük, hogy a lehető legnagyobb rekord befogadására is alkalmasak legyenek. Bár egy ilyen megvalósításnak vannak hátrányai, kezdetnek megfelelő. A fejezet végén található **getc** és **putc** változat ilyen módon működik, és a legtöbb FORTRAN-rendszerben ideiglenesen meg is felel a célnak.

Megjegyezzük, hogy az ilyen formájú bemenet nem azonos a PL/1 **stream** inputjával. A **stream** esetén a rekordhatárokról a hívó program nem kap vissza információt, bár a **skip** specifikáció segítségével lehetőség van arra, hogy karaktereket hagyjunk ki egészen a következő sorig. Ezért a **getc**-nek rekordokat kell olvasnia, hogy képes legyen a **NEWLINE**-ok jelzésére, és így mind FORTRAN-beli, mind PL/1-beli megvalósítása nagyon hasonló.

Természetesen a bemenet valóban lehet egy billentyűzetről jövő jelsorozat, a kimenet valóban egy írógépet hajthat meg, és az összes lemezfile-t is kezelhetjük ilyen formátum szerint. Ebben az esetben a **getc** és a **putc** semmitmondóvá válik. Akármi legyen is azonban a forrás vagy a kimenet, mi általában írógépszerű szöveg feltételezésével programozunk, a szükséges transzformációkat pedig bemenet esetében a lehető legkorábban, kimenet esetében pedig a lehető legkésőbb végezzük el, hogy ezáltal a kártyaolvasókat, sornyomtatókat és lemezes file-formátumokat egyformán tudjuk kezelni. (A 2. fejezet néhány példát tartalmaz erre.)

Hogyan tudnánk kipróbálni a **linecount**-ot, hogy biztosak legyünk a helyes működésben? A hibák rendszerint a program működésének a „határain", vagyis a szélső esetekben fordulnak elő. Ezek a „kivételek, amelyek erősítik a szabályt". Itt a határesetek meglehetősen egyszerűek: egy file, amelyben egy sor sincs, és egy file, amelyben egyetlen sor van. Ha a program ezeket az eseteket, valamint a több sort tartalmazó file általános esetét is helyesen kezeli, valószínű, hogy minden input esetében jól működik.

Ezért egy üres file-t adunk a **linecount**-nak. A **while**-ban levő vizsgálat „hamis" választ ad az első alkalomkor, és a törzs egyáltalán nem kerül végrehajtásra. Egyetlen sort sem számoltunk, amikor nem volt. Kitűnő.

Ha a **linecount** egyetlen sort kap, a **while**-ban levő feltétel igaz a sor minden karakterére. Az **if**-ben levő feltétel igazzá válik, amikor a **NEWLINE** érkezik, és ekkor a sort megszámláljuk. Ezután a **while** feltételvizsgálata megismétlődik, és kilépünk a ciklusból. Újra kitűnő.

A többsoros file esete: minden ugyanúgy történik, mint egyetlen sornál, csak azt figyeljük meg, hogy most minden sor után a program visszatér a **while** feltételvizsgálatához — ez a megfelelő hely a következő sor vagy az **EOF** kezelésének kezdetére. A program kiállta az ellenőrzést.

Lehet, hogy ez szőrszálhasogató részletességnek tűnik ilyen egyszerű program esetén, pedig nem az. Vannak általánosan elterjedt programozási hibák, amelyek eredményeképpen a három próba valamelyike sikertelen lehetett volna, néha úgy is, hogy a másik két próba sikeres. Meg kell tanulnunk a határesetek próbáira gondolni, amikor programunk egyes darabjait írjuk — próbáljuk ki őket gondolatban a program írása közben, majd fizikailag is a kész programon! Gyakorlatban a próbák jóval gyorsabban mennek, mint a róluk való beszéd, és csak kevés többlet-erőfeszítést igényelnek. Ez az erőfeszítés bőven megtérül!

Gyakorlat

1.3 Mi történik, ha a **linecount-nak** adott file utolsó karaktere nem **NEWLINE**? Helyesen működik-e a program? Hibás-e a viselkedése, vagy csupán természetes következménye a „sor"-ra adott definíciónknak? □

1.4 SZAVAK SZÁMLÁLÁSA

A következő számlálóprogramot a szövegfeldolgozásban alkalmazzuk — megszámlálja valamiben a szavakat. Segítségével választ kaphatunk olyan kérdésekre, mint pl. „Hány szó van ebben a könyvben?" (Kb. 80 000, a programokon kívül.) A mi számunkra a szó tetszőleges karakterek sorozata, amely nem tartalmaz szóközöket, tabulátorokat és újsor-karaktereket. Minden alkalommal számlálunk egyet, amikor a program a szón kívüli állapotból átmegy a szón belüli állapotba. Az **inword** nevű változót használjuk a program állapotának regisztrálására minden időpillanatban; kezdetben az értéke: „szón kívül vagyunk".

Ennek leírása RATFOR-ban igen tömör:

```
# wordcount — szószámlálás a standard inputon
    character getc
    character c
    integer inword, wc

    wc = 0
    inword = NO
    while (getc(c) ¬= EOF)
        if (c == BLANK | c == NEWLINE | c == TAB)
            inword = NO
```

```
        else if (inword == NO) {
            inword = YES
            wc = wc + 1
            }
    call putdec(wc, 1)
    call putc(NEWLINE)
    stop
    end
```

A **TAB** egy újabb szimbolikus konstans, amelynek értéke az a belső karakterkód, amelyet **getc** ad gépünkön a vízszintes tabulátorjelre. A függőleges vonás | a logikai *vagy* operátor. A PL/1 felhasználók számára ez ismerős; a FORTRAN programozók **.or.**-ként ismerik.

A **YES** és **NO** szimbolikus konstansok értéke valószínűleg 1 és 0, de ez nem lényeges (nem is törődünk vele), fontos, hogy különbözőek legyenek és mindenki egyetértsen az értékükben. Sokkal könnyebben olvashatók és érthetőbbek, mint a FORTRAN-ban használatos **.true.** és **.false.** logikai értékek vagy a PL/1-ben szereplő **'1'b** és **'0'b.** Ebből az okból és mivel a FORTRAN néhány önkényes megszorítást tesz a logikai összehasonlításokra, mi az **integer**-hez ragaszkodunk és a **YES**, ill. **NO**-hoz hasonló szimbolikus konstansokat használunk.

Ez a példa egyúttal több fontos RATFOR-beli vezérlési szerkezetet is tartalmaz.

1. Egy **if** utasítás tartalmazhat **else**-t olyan tevékenység leírására, amelyet akkor kell végrehajtani, ha az **if**-ben szereplő feltétel hamis:

```
if (feltétel)
    utasítás
else
    utasítás
```

Ez azt jelenti: „ha a feltétel igaz, hajtsd végre az **if**-et követő utasítást; különben hajtsd végre az **else**-t követő utasítást". Az **if—else** szerkezet végrehajtása során a két utasítás közül az egyik és csakis az egyik kerül végrehajtásra. Természetesen bármelyik utasítás meglehetősen komplikált is lehet; a **wordcount**-ban az **else** után még egy **if** áll.

2. RATFOR-ban bármelyik utasítást helyettesíthetünk egy kapcsos zárójelekben álló teljes utasításcsoporttal; ezt a szerkezetet *összetett utasításnak* nevezzük. A zárójelek a PL/1-beli **do** és **end** utasításokhoz vagy az ALGOL és származékainak **begin—end**-jéhez hasonlóan viselkednek. A zárójelekben belül levő utasításokat egyetlen utasításnak kell tekinteni. Ily módon, ha **inword** értéke **NO**, az

```
else if (inword == NO) {
    inword = YES
    wc = wc + 1
    }
```

programrészben mindkét értékadás végrehajtásra kerül. Mint minden ilyen nyelvben, a kapcsos zárójeleket elvileg tetszőleges mélységben egymásba skatulyázhatjuk.

Mi a kapcsos zárójeleket részesítjük előnyben a **begin—end** vagy **do—end** szerkezetekkel szemben, főleg mivel vizuálisan kevésbé zavarók, mint a teljes szavak, és sokkal könnyebb őket írni. A kapcsos zárójeleket következetesen pozicionáljuk: a

bal oldalit abban a sorban, amelyik az összetett utasítást vezérli, a jobb oldali pedig egyedül áll egy sorban, ugyanolyan bekezdési mélységben, mint az a csoport, amelyet befejez.

Valóban figyelemre méltó, milyen parázs vita kerekedhet olyan jelentéktelen kérdésből, hogy vajon a kapcsos zárójelek jobbak-e, mint a **begin** és az **end**, és hogy hol kell elhelyezkedniük. Ahelyett, hogy mélyebben belemennénk ebbe a vitába, elég annyit mondanunk, hogy a saját stílusunkat kényelmesnek és olvashatónak találjuk, de bárki változtathat rajta, ha azt saját programjában célszerűnek találja. Azt tanácsoljuk azonban, hogy bármilyen formai szabványt használjunk is, legyünk következetesek annak alkalmazásában.

Bár a **wordcount**-ban a **while**-t követő programrész bonyolult, logikailag mégis csak egyetlen **if—else** utasítás, nem összetett, ezért nincs szükség kapcsos zárójelekre. Ha azonban valaki szükségét érzi, elhelyezheti őket.

Az **else—if** szerkezet gyakran előfordul programjainkban, sokszor egy hosszabb **if...else if...else** lánc formájában, amely több változat közül legfeljebb egyet hajt végre. A láncok hosszabbá válnak, ha a megfelelő helyekre további **else if** utasítást helyezünk el. Ilyen láncok olvasását három szabály teszi könnyűvé:

1. Menjünk végig a feltételeken, ameddig egy olyant találunk, amely teljesül — az első ilyen választja ki a végrehajtásra kerülő változatot.
2. Ha egyik feltétel sem teljesül, akkor a lánc legvégén álló **else**-hez tartozó utasítás kerül végrehajtásra (ha van ilyen).
3. Bármelyik esetben is a végrehajtás közvetlenül az utolsó **else** törzse után folytatódik. (Vagy az utolsó **else if** törzse után, ha a lánc végén nem **else** áll.)

Néhány nyelvben a „**case**" utasítást használják a többutas döntés közvetlen kifejezésére. Egy **case** nélküli nyelvben rendszerint az **else if** lánc a legjobb helyettesítő megoldás. Nyomatékosan ajánljuk a többutas döntéseknek ilyen formájú leírását a tetszőlegesen skatulyázott **if—else** fák helyett, mivel ez kevésbé zavaró jellegű. Az összes **else if**-et azonos bekezdési mélységben írjuk, hangsúlyozva ezzel, hogy a szerkezet valóban többutas döntés.

Hogyan próbáljuk ki a **wordcount**-ot? Ezt a tevékenységet a legjobb akkor kezdeni, amikor a programot írjuk. Egy program helyességének fő biztosítéka az a szellemi erőfeszítés, amit kifejtünk azért, hogy első alkalomra jó legyen. A **wordcount**-ot két állapotra alapozott algoritmussal írtuk — szón belül vagyunk és szón kívül vagyunk. Ha helyesen oldjuk meg az állapotok közötti átmeneteket, a kezdeti feltételeket jól állítjuk be, és minden új szó esetén megfelelően számoljuk az átmeneteket, akkor megbízhatunk a programban. A tesztelés természetesen továbbra is szükséges ahhoz, hogy ellenőrizzük az algoritmus helyességét és azt, hogy a program jól valósítja-e meg.

Már egy kevéssé bonyolult program esetében sem tudjuk kipróbálni az összes lehetséges bemenő adatokat. Amint már említettük, a programozók keserű tapasztalatok árán tanulták meg, hogy a program határesetei azok a helyek, amelyeket legérdemesebb megvizsgálni, mivel a hibák leggyakrabban itt jelentkeznek. Emellett, ha ki tudjuk mutatni, hogy egy program helyesen dolgozik a határeseteiben, akkor nagy valószínűséggel a határesetek között is mindenhol hibátlan lesz. Így a határesetekre kiélezett kritikus tesztek egy kis készlete többet ér, mint a vaktában végzett próbák hosszú sorozata.

Bár erre nehéz pontos definíciót adni, ösztönszerűen „határeset"-nek érzünk egy olyan adatot, amelyre a program kénytelen a szomszédos értéktől különböző módon reagálni. Például az „üres input" esete a legtöbb program számára határeset. A **wordcount** tesztelésekor, ha nincs input, a **getc** első hívása **EOF**-et ad vissza, a **while**

törzse nem hajtódik végre, a szószámlálóban zérust kapunk, amint az várható volt. Hasonló határeset, amikor input van, de szavak nincsenek: érdemes ezt is kipróbálni! Ha a bemenet kizárólag szóközökből, tabulátorokból és újsor-karakterekből áll, az első **if** mindig teljesül, így **wc** soha nem növekedik; az eredmény újra zérus, ahogy annak lennie is kell.

Célszerű azt is ellenőrizni, hogyan dolgozik a **wordcount**, amikor az inputban egyetlen szó van, tekintet nélkül arra, hogy ez melyik helyen jelenik meg, valamint azt, amikor két szó van az inputban, különböző helyeken. Ha ezek a tesztek nem utalnak hibára, meglehetősen bízhatunk abban, hogy a program helyes.

Egy utolsó megjegyzés. A **wordcount** kipróbálásakor nyilvánvaló volt, hogy az egyes bemenetekhez milyen kimenet tartozik. Nagyobb programok esetében ez nem mindig ilyen nyilvánvaló. Mégis, a tesztelés egyik alapelve, hogy előre tudnunk kell, milyen választ fogunk kapni az egyes esetekben. Ha ezt nem tudjuk, akkor nem tesztelünk, hanem kísérletezünk. A program írásával velejáró kötelesség tehát egy átfogó teszt-adatkészletnek, valamint olyan eredményeknek az előkészítése, amelyekkel a futás eredményeit összehasonlíthatjuk.

Gyakorlatok

1.4 Kombináljuk a **charcount, linecount** és **wordcount** tevékenységet egyetlen programmá! Milyen sorrendben kellene a három számláló tartalmát kinyomtatni? Mi a jobb: ha egy programunk van, amely három dolgot csinál vagy három programunk, amelynek mindegyike egyet intéz el? □

1.5 Módosítsuk a **wordcount**-ot úgy, hogy mondatokat is számoljon! (Alkossunk saját definíciót a mondatra!) Várható, hogy gyakran fogják használni ezt a programot? □

1.5 TABULÁTOROK ELTÁVOLÍTÁSA

Tegyük fel, hogy horizontális tabulátorkaraktereket tartalmazó szöveges file-t akarunk listázni olyan eszközön, amelyik nem ismeri a tabulátorokat. Első közelítésként megelégedhetünk rögzített tabulátorpozíciókkal, mondjuk, minden nyolcadik oszlopban. Ebben az esetben egy tabulátorkaraktert egytől nyolcig terjedő számú szóközzel kell helyettesíteni. Írjuk meg a **detab** nevű programot ennek végrehajtására!

A program szerkezete azonos a **copy**-éval, kivéve, hogy a **while** ciklust át kell dolgoznunk:

```
while (getc(c) ¬= EOF)
    if (c == TAB)
        szóközök kivitele a köv. tab-ig
    else
        a c-ben levő karakter kivitele
```

Honnan tudjuk, hogy mikor értünk a következő tabulátorpozícióhoz? Egyik lehetőség, hogy beépítjük a főprogramba azt az információt, hogy a tabulátorok minden nyolcadik oszlopban helyezkednek el; ezután egy aritmetikai vizsgálat segítségével eldönthető, hogy az aktuális oszlop tabulátorpozíció-e. Ezzel a megoldással akkor van baj, ha meg akarjuk változtatni a programot, mondjuk úgy, hogy a tabulátorokat olyan pozíciókban engedjük meg, amelyeket nem lehet egy egyszerű arit-

metikai formula segítségével kapcsolatba hozni egymással. Ha a „minden nyolcadik oszlop" döntést alapvetően beépítjük a programba, később már nehéz lesz onnan kiküszöbölni.

Rugalmasabb megoldás, ha bevezetjük a tabulátorpozíciók egy tömbjét, amelyet jelenleg a „minden nyolcadik oszlop" szabály alapján látunk el kezdőértékkel. Ezt sokkal könnyebb változtatni; sőt még azt sem mondtuk meg, hogy vajon a tömb a tabulátorkarakterek listáját tartalmazza-e vagy egy **YES/NO** jelzést minden egyes oszlopra, ahogyan azt az írógépeknél teszik. A tabulátorkarakterek egy tömbben való ábrázolása mindenképpen olyan programot eredményez, amelyet könnyen lehet az általánosabb alkalmazások irányában továbbfejleszteni.

Bármelyik utat választjuk is, mindig érdemes egy **tabpos** függvényt bevezetni, amely közli a főprogrammal, hogy egy adott oszlop tabulátorpozíció-e vagy sem. Ily módon elkerüljük a vezérlőciklus alapvető logikájának tabulátorpozíció-számításokkal való megzavarását, és elrejtjük a tabulátorpozíciók reprezentációját a főprogram elől.

Nyilvánvaló, hogy a programnak számon kell tartania, melyik oszlopnál jár, és fel kell ismernie a szövegsorok végét, hogy vissza tudja állítani az oszlopszámlálót. A tabulátoreltávolító második közelítése a következő:

```
tab-ok kezdeti beállítása
col = 1
while (getc(c) ¬= EOF)
    if (c == TAB)
        szóközök kivitele és col növelése
        until (tabpos(col, tabs) == YES)
    else if (c == NEWLINE) {
        call putc(c)
        col = 1
        }
    else {
        call putc(c)
        col = col + 1
        }
```

Ebben a programban egy **else**-vel végződő **else if** lánc van, ahol az utolsó **else** intézi el a „bármi más" esetet, vagyis azt, ami nem **TAB** és nem is **NEWLINE**.

A **tabpos** **YES**-t ad vissza, ha a **col** sorszámú oszlop tabulátorpozíció, és **NO**-t különben. Elvileg ez egyszerű feladat. De álljunk meg egy percre, és gondoljunk a határesetekre vonatkozó eszmefuttatásunkra! Egyik nyilvánvaló határeset az utolsó tabulátorpozíció. Mi történik, ha a bemenetben egy tabulátorkarakter fordul elő az utolsó tabulátorpozíciót követő egyik oszlopban?

Egyik megoldás, hogy megtiltjuk a tabulátorkarakterek használatát egy bizonyos oszlopindex után, ám eleve kudarcra ítélt vállalkozás olyan programot írni, amely vakon feltételezi, hogy hibátlan adatokat kap. Az is elképzelhető, hogy a **detab** leáll és/vagy hibaüzenetet ad, de ez aligha kívánatos egy általános célú eszköznél. Miért ne tennénk inkább valami értelmesebbet? Egy program adjon elfogadható eredményeket, ha elfogadható adatokat kap, és semmi elfogadhatatlan sincs abban, hogy a szöveg sok tabulátorkaraktert tartalmaz. Építsük fel a **detab**-ot úgy, hogy ha az utolsó tabulátorpozíció után tabulátorkarakter fordul elő, akkor azt egyetlen szóközzé konvertáljuk.

Mivel a

```
szóközök kivitele és col növelése
until (tabpos (col, tabs)==YES)
```

ciklus akkor fejeződik csak be, amikor éppen egy tabulátorpozícióhoz ér, biztosítanunk kell, hogy ez még akkor is bekövetkezzék, ha a sor túlnyúlik az utolsó tabulátorpozíción. Erre biztos módszer annak feltételezése, hogy az utolsó explicit tabulátorpozíció után *minden* oszlop tabulátorpozíciónak számít. Ezt a **tabpos**-nak kell megoldania.

Már csak néhány részlet van hátra, és megírhatjuk a **tabpos**-t. Azt a megoldást választottuk, hogy a **tabs** nevű tömb elemei **YES**-t tartalmaznak, ha ebben az oszlopban van tabulátorpozíció, **NO**-t, ha nincs. A végső RATFOR változat a következő:

```
# detab — tab-ok konvertálása megfelelő számú szóközre
    character getc
    character c
    integer tabpos
    integer col, i, tabs(MAXLINE)

    call settab(tabs)       # tab-ok kezdeti beállítása
    col = 1
    while (getc(c) ¬= EOF)
        if (c == TAB)
          { repeat {
                call putc(BLANK)
                col = col + 1
                } until (tabpos(col, tabs) == YES) }
        else if (c == NEWLINE) {
            call putc(NEWLINE
            col = 1
            }
        else {
            call putc(c)
            col = col + 1
            }
    stop
    end

# tabpos — igaz, ha col tabulátorpozíció
    integer function tabpos(col, tabs)
    integer col, i, tabs(MAXLINE)

    if (col > MAXLINE)
        tabpos = YES
    else
        tabpos = tabs(col)
    return
    end
```

A **detab** a **tabs** tömb kezdő beállítását a **settab**-bal végzi el, a **tabpos** által elvárt adatábrázolásnak megfelelően.

```
# settab — tab-ok kezdeti beállítása
   subroutine settab(tabs)
   integer mod
   integer i, tabs(MAXLINE)

   for (i = 1; i <= MAXLINE; i = i + 1)
        if (mod(i, 8) == 1)
             tabs(i) = YES
        else
             tabs(i) = NO
   return
   end
```

A **mod** függvény azt a maradékot adja eredményül, amely első argumentumának a másodikkal való osztásakor lép fel. Ebben az esetben **mod (i, 8)** az 1, 2, 3, 4, 5, 6, 7, 0, 1, 2, ... sorozatot állítja elő, *i* növekedésének megfelelően. Ily módon minden nyolcadik alkalommal keletkezik egy 1-es, és ezt használjuk a tabulátorpozíciók jelölésére. Ez a **mod** függvény egyik szokványos felhasználása.

Értelmetlennek tűnhet egy ötsoros függvény megírása, amelyet csak egyszer hívunk, azonban a **settab** célja az, hogy elkülönítse az adatábrázolást attól a rutintól, amelynek azt nem kell ismerni. Egy **detab** méretű program esetében ez nem feltétlenül szükséges, de nagyobb programoknál már alapvető fontosságú, hogy azokat kisebb részekre bontsuk, amelyek jól körülhatárolt illeszkedési szabályok szerint kommunikálnak egymással. Minél kevesebbet tud a program egyik része arról, hogyan működik a másik rész, annál könnyebb bármelyiket megváltoztatni.

A használatban levő programok többségét alkalmazásuk során állandóan változtatják és tökéletesítik, és sok programozó ideje legnagyobb részét ezeknek az „élő" programoknak a karbantartásával tölti. Ez igen költséges munka, ezért a programok tervezésekor az egyik legfontosabb szempont, hogy az könnyen módosítható legyen.

Ennek elérésére az általunk ismert legjobb módszer az, hogy a programok egyes részei már a megíráskor annyira lazán kapcsolódjanak, amennyire ez csak lehetséges, így az egyik megváltoztatása nem hat ki a többire. Megpróbáljuk külön részekben elhelyezni azokat a részleteket, amelyek a program általánosságát és működésének módját korlátozzák. Például a **getc** és **putc** elkülönítik a jelkészlet, sorok, rekordok, file-kijelölések és file-vége kezelés minden részletét. Hasonlóképpen, a **detab**-ot úgy szerveztük, hogy a főprogram nem foglalkozik a tabulátorpozíciók ábrázolásával, csupán az oszlopok számlálásával.

A feladat földarabolása azt is elősegíti, hogy a tervezésnek egyszerre csupán egy szempontjára összpontosítsuk figyelmünket. Nagyobb az esélyünk arra, hogy egy jó és érthető **detab**-ot készítünk, ha a feladatát az oszlopszámlálásra korlátozzuk. Ugyancsak valószínűbb, hogy a **tabpos** helyes lesz, ha csupán a tabulátorpozíciókat kezeli, és egy könnyen megjegyezhető specifikációjú függvényt valósít meg. A legjobb programok egymáshoz lazán kapcsolódó függvényekből épülnek fel, amelyek mindegyike egy-egy egyszerű feladatot végez el.

A **detab** két új vezérlési szerkezetet is bemutat, amelyekkel a készletünk majdnem teljessé vált. A **repeat—until** olyan ciklus, amely egyszer vagy többször addig ismétlődik, ameddig a végén álló feltétel nem teljesül. Ez eltér a **while**-tól, amelyik a ciklus elején vizsgálja a feltételt, mielőtt bármi is végrehajtódnék. A **while** gyakrab-

ban alkalmazhatónak tűnhet, de mindkét formának megvan a maga alkalmazási területe. A **detab**-ban a **repeat**-re van szükség, mivel biztosítanunk kell, hogy minden bejövő tabulátorkarakter legalább egy szóköz kivitelét eredményezze.

A **for** utasítás tömören foglalja össze a következő gyakori szerkezetet:

```
for (kezdőérték; feltétel; léptetés)
    törzs
```

Ez a következővel ekvivalens:

```
{
kezdőérték-beállítás
while (feltétel) {
        törzs
        léptetés
        }
}
```

A **for** utasítás alapgondolata a ciklusvezérlő információnak egy sorban való tömörítése, ami hasonló a konvencionális FORTRAN-beli **do** utasításhoz. Ettől eltérően azonban a **for** utasítás nincs számtani sorozatok előállítására korlátozva. Ez fontos, mivel amint látni fogjuk, a ciklusok túlnyomó többsége nem FORTRAN **do**, még csak nem is PL/1 **do**, hanem bonyolultabb vizsgálatot, általánosabb kezdőérték-beállítást és ciklusváltozó-módosítást igényel. A kezdőérték-beállítás és a ciklusváltozó módosítása bármilyen FORTRAN utasítás lehet; nincs az **i=1**, ill. **i=i+1** esetre korlátozva.

Gyakorlatok

1.6 Fejtsük ki a **settab**-ban levő **for** utasítást a fenti formula alapján! Lehet-e azt egy **do**-ciklussal helyettesíteni? □

1.7 Teszteljük a **detab**-ot! Ki kell állnia minden próbát, amelyet a **copy** kiáll. (Kivéve, hogy a tabulátorkaraktereket 1-től 8-ig terjedő számú szóközzel helyettesíti.) Ezen túlmenően több határeset tartozik a tabulátorpozíciókhoz. Figyelembe vehetünk olyan, csupa **x** betűből álló sorokat, amelyek mindegyikében egy-egy tabulátorkarakter van elhelyezve, különböző pozícióban. Mi történik, ha az utolsó tabulátorpozíció után tabulátorkarakter fordul elő? □

1.8 Mit csinál a **detab**, ha az input egy visszaléptető (backspace) karaktert tartalmaz? Módosítsuk úgy, hogy helyesen működjön! □

1.9 Nyilvánvaló, hogy a **detab**-ot több más módon is meg lehet írni. Készítsük el a következő változatokat és hasonlítsuk össze méretük, bonyolultságuk és későbbi módosíthatóságuk alapján!

a) Minden nyolcadik pozíció tabulátor; a **tabs** tömböt nem használjuk.

b) A **tabs** tömb azon oszlopok listáját tartalmazza, amelyeken tabulátor van; a listát egy 0 érték zárja le.

c) A **tabs** tömbben minden elem a következő tabulátorig terjedő oszlopok számát tartalmazza; az utolsó elem nulla.

d) Ismételjük meg *b)*-t és *c)*-t úgy, hogy végjel helyett a tabulátorpozíciók számát adjuk meg! □

1.6 A PROGRAM KÉZI FORDÍTÁSA

A **detab**-nak FORTRAN-ra való fordításából kitűnik a RATFOR-ban való programozás néhány előnye:

```
c detab — tab-ok konvertálása szóközökre; FORTRAN változat

      integer getc
      integer c
      integer tabpos
      integer col, i, tabs(MAXLINE)

c
c     tab-ok kezdeti beállítása

      call settab(tabs)
      col =1
10    if (getc(c) .eq. EOF) goto 60
         if (c .ne. TAB) goto 30
20          call putc(BLANK)
            col = col + 1
            if (tabpos(col, tabs) .ne. YES) goto 20
            goto 50

c        különben, ha

30       if (c .ne. NEWLINE) goto 40
            call putc(NEWLINE)
            col = 1
            goto 50

c        különben

40          call putc(c)
            col = col + 1
50       goto 10
60    stop
      end

c settab — tab-ok kezdeti beállítása; FORTRAN változat

      subroutine settab(tabs)
      integer mod
      integer i, tabs(MAXLINE)
```

```
c
      i = 1
10    if (i .gt. MAXLINE) goto 20
         if (mod(i, 8) .eq. 1) tabs(i) = YES
         if (mod(i, 8) .ne. 1) tabs(i) = NO
         i = i + 1
         goto 10
20    return
      end

c tabpos — igaz, ha col tabulátorpozíció; FORTRAN változat

      integer function tabpos(col, tabs)
      integer col, i, tabs(MAXLINE)
c
      if (col .gt. MAXLINE) tabpos = YES
      if (col .le. MAXLINE) tabpos = tabs(col)
      return
      end
```

Amint látjuk, nehezebb a vezérlési struktúrát felismerni, még az azonos bekezdési mélységek és a pszeudokód formájában adott magyarázatok ellenére is. Az utasításszámok és a **goto**-k elhomályosítják a program jelentését.

A FORTRAN-program írása teljesen mechanikus művelet, ha a RATFOR szöveg már készen van. Minden RATFOR szerkezetből egyszerű FORTRAN szerkezet lesz; a fordításhoz nincs szükség találékonyságra. Valójában, a **goto** bármely ötletes használata a végső programban, bármely alkalmazás, amely nem alkot könnyen felismerhető alakzatot, csak gondot okoz annak, akinek a programot meg kell értenie, beleértve magát a program íróját is.

Mostanra már az olvasónak valószínűleg feltűnt, hogy ez a fordítás nem elég hatékony. Vannak benne **goto**-ra mutató **goto**-k, valamint többszörösen végrehajtott vizsgálatok, mint például:

```
if (col .gt. MAXLINE) tabpos = YES
if (col .le. MAXLINE) tabpos = tabs(col)
```

a FORTRAN-ban szokásos

```
tabpos = YES
if (col .le. MAXLINE) tabpos = tabs(col)
```

programozási forma helyett. Ezen kívül a FORTRAN-ban használható a **do** utasítás a kezdőérték-beállításra, a léptetésre és a ciklusváltozó ellenőrzésére. Miért nem használjuk ezeket a **settab**-ban?

Nincsen semmi kifogásunk a **do** ellen, ahol a fordítás nyilvánvaló, sőt a RATFOR tartalmaz is egy **do** utasítást (amelyet úgy kapunk, hogy a **do** utáni utasításszámot elhagyjuk). A **settab**-ban levő kezdeti beállítások ciklusa könnyű eset, mivel **MAXLINE** mindig pozitív. De felhívjuk a figyelmet arra, hogy a FORTRAN **do**-ja erősen korlátozott: az indexnek valamilyen pozitív értéktől valamilyen pozitív határig kell változnia és a FORTRAN sok változatában a ciklus legalább egyszer végigfut a

határoktól függetlenül, mivel a vizsgálatot a ciklus végén végzik el. Óriási előnye a **while**-nak és a **for**-nak, hogy a vizsgálatnak a ciklus elején való elvégzésével automatikusan ellenőrzik azt a speciális esetet, amikor semmit sem kell csinálni, és ilyenkor nem is tesznek semmit.

Továbbá, ha **do**-kban gondolkozunk **while**-ok helyett, erősödik az az érzésünk, hogy valamit számlálnunk kell, vagy hogy a ciklusváltozónak *m* lépésenként 1-től *n*-ig kell futnia. De miért van szükség mindig ciklusváltozóra? Még ha sok fogás létezik is tetszőleges sorozatoknak számtani sorozatokra való leképzésére, ne felejtsük el, hogy ezek *fogások!* Ezek elhomályosítják a program logikáját, és növelik a hibák elkövetésének esélyét. Amint ezt látni fogjuk a 9. fejezetben található mérésekből, ciklusokra ritkán alkalmazható természetes módon a **do** által előírt forma.

Felvethető az is, hogy a **do** hatékonyabb, mint az **if**-ekből és **goto**-kból konstruált **for**. Végtére is, a fordítóprogram ismeri a **do** ciklust és feltehetően igyekszik jól lefordítani. Ez néhány esetben igaz; a mátrixszámításokban például gyakran ki tudják használni a fordítóprogramoknak a **do** kezelésére vonatkozó gondosságát. De a **settab** hatékonysága mondvacsinált probléma; a rutin csak egyszer kerül hívásra. Az elágazásokra való ugrás és a többszörösen végrehajtott vizsgálatok a **tabpos**-ban és a **settab**-ban szintén lényegtelenek. A futási időre gyakorolt hatásuk kimutathatatlanul kicsi.

A program időfelhasználásával kapcsolatos mérések azt mutatták, hogy a **detab** idejének 60%-át alacsony szintű inputtal (a **getc** által hívott rutinokban), további 20—25%-át alacsony szintű outputtal (a **putc** által hívott rutinokban) tölti. Tehát, ha a program megmaradt részét úgy tudnánk átírni, hogy zérus időt igényeljen (ami aligha valószínű), a gyorsítás csak 15—20% volna.

A tanulság nyilvánvaló: egy program egyszerű, kézenfekvő megírásával elérjük azt, hogy helyesen működjön, és a minimumra csökkenjen a zavarosság veszélye. Ezután teljesítményméréseket végezhetünk, hogy eldöntsük, elég jól működik-e, és ha nem, mire kell a figyelmünket összpontosítani. Egy adott algoritmusnál a sebességet szinte minden esetben csak az olvashatóság rovására növelhetjük. Csak akkor áldozzuk fel az áttekinthetőséget a sebességért, ha tudjuk, hogy a helyes feladatot oldjuk meg helyesen és tudjuk, hogy ez megéri az áldozatot.

PL/1-ben a program a következő:

```
/* detab — tab-ok konvertálása szóközökre */
    detab: procedure options (main);
    declare getc entry (fixed binary) returns (fixed binary);
    declare putc entry (fixed binary);
    declare c fixed binary;
    declare settab entry ((*)fixed binary);
    declare tabpos entry (fixed binary, (*)fixed binary) returns (fixed
        binary);
    declare (col, tabs(MAXLINE)) fixed binary;

    call settab(tabs);      /* tab-ok kezdeti beállítása */
    col = 1;
    do while (getc(c) ¬= EOF);
        if c = TAB then do;
          loop:
            call putc(BLANK);
            col = col + 1;
```

```
            if tabpos(col, tabs) ¬= YES then
                goto loop;
            end;
        else if c = NEWLINE then do;
            call putc(NEWLINE);
            col = 1;
            end;
        else do;
            call putc(c);
            col = col + 1;
            end;
        end;
    end detab;

/* settab — tab-ok kezdeti beállítása */
    settab: procedure (tabs);
    declare (i, tabs(*)) fixed binary;
    do i = 1 to MAXLINE;
        if mod(i, 8) = 1 then
            tabs(i) = YES;
        else
            tabs(i) = NO;
        end;
    end settab;

/* tabpos — igaz, ha col tabulátorpozíció */
    tabpos: procedure (col, tabs) returns (fixed binary);
    declare (col, tabs(*)) fixed binary;
    if col > MAXLINE then
        return(YES);
    else
        return(tabs(col));
    end tabpos;
```

Ez alkalommal **do** utasítást használtunk a fordításban, mivel a PL/1 **do**-ja jobban hasonlít a **while**-hoz, mint a FORTRAN-é. Speciálisan a PL/1 **do** egyáltalán nem kerül végrehajtásra, ha a ciklusváltozó induláskor átlépi a felső határt. PL/1-ben nincs meg a **repeat—until** megfelelője, így fel kellett azt építenünk egy **if**-fel és egy **goto**-val, nagyjából úgy, mint a FORTRAN-ban.

Gyakorlatok

1.10 Írjuk meg az **if**, az **if—else**, a **while**, a **for** és a **repeat—until** fordítását FORTRAN-ban! Csak logikai **if**-et és **goto**-kat használjunk! Alkalmazzuk szabályainkat a **detab** RATFOR-programjára, és hasonlítsuk össze eredményünket a FORTRAN változatunkkal! Mit mondhatunk az előállított **goto**-k „optimalizálásáról"? Milyen jól dolgozik FORTRAN fordítóprogramunk? □

1.11 Írjuk meg az **if**, **if—else**, **while**, **for** és **repeat—until** PL/1-re való fordításának szabályait! Csak akkor használjunk **goto**-t, ha ez elkerülhetetlen! Alkalmazzuk szabályainkat a fenti RATFOR-programra, és hasonlítsuk össze az eredményt a fenti PL/1 változattal! Mit mondhatunk a **while** és **for** ciklusok fordításának „optimalizálásáról"? □

1.7 NÉHÁNY SZÓ A STRUKTURÁLT PROGRAMOZÁSRÓL

Az olvasó csodálkozhat, hogy miért foglalkozunk olyan sokat az **if—else** és a **while** használatával az **if** és a **goto** helyett. Végül is, az utóbbiakkal mindent, sőt még többet is meg lehet oldani mint a RATFOR vezérlő utasításaival. Kiderül azonban, hogy a **goto**-k által biztosított többletszabadság pontosan az, amit nem akarunk. Ha a vezérlő utasítások egy szűk készletére korlátozzuk magunkat, programunk jobban átgondolt, könnyebben olvasható, és ezért kisebb benne a hibalehetőség. Ezek az előnyök messze meghaladják mindazt a kifejezésbeli szabadságot, amelyet esetleg feláldozunk.

A legszűkebb értelemben ez az, amit „strukturált programozás"-nak nevezünk: programozás egy korlátozott vezérlőutasítás-készlet segítségével. E készlet tartalmazza az utasítások csoportosítását a RATFOR kapcsos zárójeleihez hasonló konstrukcióban; az **if—else** szerkezettel kifejezett döntéseket, a **while**-hoz, **for**-hoz vagy **repeat—until**-hoz hasonló ciklusokat, valamint a szubrutin- és függvényhívásokat. Egyesek néha azt állítják, hogy pusztán e szerkezetek használatával természetesen és automatikusan jó programok jönnek létre. Ez nem igaz, mert mechanikus szabályok sohasem pótolják a gondolat tisztaságát. Annyi igaz, hogy egy jól strukturált program általában csak ezeket a szerkezeteket használja építőkövekként.

A strukturált programozást az alapokhoz való ragaszkodáson keresztül közelítjük meg. Több erőfeszítést fektetünk a tervezésbe és a kódolásba (ami szórakozás), hogy csökkentsük a jóval költségesebb tesztelést és hibajavítást (ami nem az). Különös hangsúlyt fektetünk a tiszta, érthető programra, és a hatékonysági meggondolásokat későbbre halasztjuk. Ellenőrizzük és kipróbáljuk a programot már írás közben, ahelyett, hogy egy végső „hibajavító orgiában" bíznánk, amely mindent megold. Kiterjedten használjuk a szubrutinhívásokat és az utasítások csoportosítását a program modularizálásának érdekében, és a vezérlési részeket mindig **if—else** szerkezettel, valamint a ciklusutasításokkal oldjuk meg.

Mindemellett a jó programok írására elsődleges eszközünk az, hogy igyekszünk őket olvashatóvá tenni. Tapasztalatunk szerint a program minőségének legjobb kritériuma az olvashatóság: ha egy program könnyen olvasható, valószínűleg jó; ha nehezen olvasható, valószínűleg nem jó. Időről időre — amikor csak erre megfelelő példa adódik — felhívjuk a figyelmet a stílusra. Reméljük, hogy az olvasó tanulni fog ezekből a megfigyelésekből, és minden leírt programsornál szem előtt tartja az olvashatóságot.

1.8 A RATFOR ÖSSZEFOGLALÁSA

Már megismerkedtünk a RATFOR nagy részével; amint szükség lesz rá, bemutatjuk a többit is, és minden esetben utalunk arra, hogyan lehet a programot FORTRAN-ra és PL/1-re fordítani.

Segítségként e helyen közöljük a RATFOR összefoglalását. A RATFOR lényegében FORTRAN, eltekintve attól, hogy új utasításokkal egészült ki a programvezérlés megkönnyítése érdekében. Ezeken kívül néhány jelölési konvenciót is tartalmaz, hogy a programok olvashatóságát elősegítse.

A következőkben az *utasítás* bármilyen általánosan használt FORTRAN-utasítást jelöl: értékadás, deklaráció, szubrutinhívás stb., vagy a felsorolt RATFOR-

utasítások bármelyikét. Bármelyik FORTRAN- vagy RATFOR-utasítás, ill. ezekből alkotott csoport a { } kapcsos zárójelekbe tehető, és ezáltal összetett utasítássá válik, amely egyetlen utasítással egyenértékű, és mindenhol használható, ahol egy magában álló utasítás előfordulhat.

Az **if** *utasítás:*

```
if (feltétel)
    utasítás1
else
    utasítás2
```

Ha a feltétel igaz, *utasítás1*, különben *utasítás2* hajtódik végre. Az **else** rész elhagyható. Mint a legtöbb nyelvben, az

```
if (feltétel1)
    if (feltétel2)
        utasítás1
    else
        utasítás2
```

szerkezet nem egyértelmű, mivel az **else** bármelyik **if**-hez tartozhat. A kívánt értelmezés elérése céljából kapcsos zárójeleket használhatunk. Kapcsos zárójelek hiányában mindegyik **else** az őt megelőző olyan **if**-hez tartozik, amely még nem rendelkezik **else**-párral, akárcsak a PL/1-ben és egyes kis nyelvekben. A fenti példában ezen kapcsolódási szabály szerint állítottuk be a bekezdéseket, de a jövőben mindig kapcsos zárójeleket is használunk, hogy szándékunk teljesen világos legyen.

A **while** *utasítás:*

```
while (feltétel)
    utasítás
```

Ha a *feltétel* igaz, végrehajtódik az *utasítás*, majd a *feltétel* újra ellenőrzésre kerül. Ha a *feltétel* egyszer hamissá válik, a program végrehajtása a **while** törzse után következő utasításnál folytatódik.

A **for** *utasítás:*

```
for (kezdőérték-beállítás, feltétel, léptetés)
    utasítás
```

Ez egyenértékű a

```
{
kezdőérték-beállítás
while (feltétel) {
    utasítás
    léptetés
    }
}
```

szerkezettel, eltekintve az alábbi kivételtől. A *kezdőérték-beállítás* és a *léptetés* egy-egy FORTRAN utasítás. Ha akár a *kezdőérték-beállítás,* akár a *léptetés* hiányzik, a kifejtés megfelelő tagja elhagyható. Ha a *feltétel* hiányzik, akkor azt mindig igaznak kell tekinteni, ami „végtelen" ciklust eredményez.

Bár a *kezdőérték-beállítás* és a *léptetés* bármilyen FORTRAN-utasítás lehet, célszerű, ha azok a ciklus vezérlésére vonatkoznak. Ha két vagy több rész hiányzik a **for**-ból, az utasítást rendszerint célszerű **while**-lal vagy **repeat**-tel helyettesíteni.

A **repeat—until** *utasítás:*

> **repeat**
> *utasítás*
> **until** *(feltétel)*

Az *utasítás* egyszer vagy többször addig fut, amíg a *feltétel* igazzá nem válik, amikor is a ciklusból kilépünk. Az **until** rész elhagyható; ha hiányzik, akkor az eredmény végtelen ciklus, amelyből valahogy másképpen kell kilépni.

A **break** *utasítás:*

> **break**

A **break** utasítás lehetőséget nyújt egy végtelen ciklusból való kilépésre. Eredményeként azonnal kilépünk abból a ciklusból, amelyikben a **break** utasítás van (ami lehet **while, for** vagy **do**). A program végrehajtása a ciklust követő utasításnál folytatódik. Csak egyetlen ciklus fejeződik be **break** utasítással, még akkor is, ha a **break** többszörösen egymásba skatulyázott ciklusok belsejében van elhelyezve.

Eddig még nem használtuk a **break**-et; ezt a 2. fejezetben tesszük majd meg. Létezik két további RATFOR-utasítás is, amelyeket egyáltalán nem használunk ebben a könyvben, de amelyek néha a legtisztább módon fejeznek ki egy-egy műveletet. Egyikük a **break**-hoz hasonló.

A **next** *utasítás:*

> **next**

A **next** utasítás hatására az őt tartalmazó ciklus azonnal áttér a következő iterációra, kihagyván a ciklustörzs hátralevő részét. A **while, do** vagy **until** esetén a vezérlés a feltételrészre kerül, végtelen **repeat** esetén a ciklus elejére, **for** esetén pedig a léptetésre. (Ez az oka annak, hogy a **for**-t nem lehet közvetlenül egy **while** ciklussá kifejteni.)

A **do** *utasítás:*

> **do** *határok*
> *utasítás*

Ez egy szabványos FORTRAN **do** ciklust hoz létre. A *határok* megfelelnek egy szabályos FORTRAN **do** specifikációnak, mint **i=1, n.**

A RATFOR utasítások egy soron belül akárhol elhelyezkedhetnek. Lényeges a bekezdések következetes alkalmazása, mivel ebből látható, hogy melyik utasítás melyiket vezérli. Általában a sor vége jelöli egy utasításnak a végét, de az

```
if (c == NEWLINE)
    nl = nl + 1
```

jellegű szerkezet nyilvánvalóan nem fejeződik be az **if**-et tartalmazó sor végén, hanem értelemszerűen folytatódik. Ugyanez áll olyan feltételekre, amelyek egynél több sorban helyezkednek el, pl.

```
while (c == BLANK
     | c == TAB
     | c == NEWLINE)
       i = i + 1
```

A vesszővel végződő sorok szintén folytatódnak.

Egy gyakori kérdésre válaszolva: egy tetszőleges FORTRAN-program nem szükségszerűen RATFOR-program. RATFOR-ban a szóközöknek jelentőségük van az olyan kulcsszavakban, mint az **if**, a szimbolikus konstansokban, mint a **NEWLINE** és a relációjelekben, mint a >=. Ezek nem tartalmazhatnak szóközöket, mert ez megakadályozza felismerésüket. Továbbá a kulcsszavak foglaltak, nem használhatók változók neveiként. A szabványos FORTRAN magyarázatai (kommentárjai), folytatási konvenciói és aritmetikai **if** utasítása nem kompatibilisak a RATFOR-ral, de mivel a RATFOR mindegyik helyett célszerűbb megoldást kínál, ez nem okoz komoly nehézséget.

Másrészről, a RATFOR nem áll messze a FORTRAN-tól. Programjainkban a sorok mintegy 20%-át a RATFOR vezérlő utasításai teszik ki, mint az **if** és **else**; minden más utasítás valódi FORTRAN. Törekedtünk a szabványos (ANSI) FORTRAN egy olyan részhalmazához ragaszkodni, ami a különböző gépek között változtatás nélkül átvihető. Összes programunkat átfuttattuk egy ellenőrző programon, amely megvizsgálta, hogy nem tértünk-e el ettől a portábilis részhalmaztól. Továbbá, hogy a portabilitási követelményeinket ellenőrizzük, a nagyobb programokat három gépen futtattuk le, és a többieket is legalább kettőn, a szó szoros értelmében változtatás nélkül.

Mivel ragaszkodtunk a portabilitáshoz, néhány szerkezet furcsának tűnhet. Pl. az ANSI* FORTRAN megtiltja a bonyolult indexeket, így esetenként közbenső változókat kellett bevezetnünk ezek elkerülésére. Hasonlóképpen a változók nevei (de nem a szimbolikus konstansoké) legfeljebb hat karakter hosszúságúak lehetnek. Ezért néhány változónév meglehetősen erőltetett, amiért előre elnézést kérünk.

1.9 A GETC ÉS PUTC EGYSZERŰ VÁLTOZATAI

A **getc** és a **putc** elég fontosak ahhoz, hogy a rendelkezésre álló operációs rendszer tartalmazza őket, de ez gyakran még sincs így. Ez azonban nem jelenti azt, hogy nélkülük kell boldogulnunk, mivel igen könnyű olyan változatokat előállítani, amelyek sok célt kielégítenek. Például a következő **getc** változat egy tipikus FORTRAN környezetbe illik, és a bemenetét kártyáról veszi. Elolvasásával működése teljesen világossá válik.

* ANSI (American National Standard Institute = az Egyesült Államok Szabványügyi Hivatala).

```
# getc (egyszerű változat) — karakterolvasás a standard inputról
   character function getc(c)
   character buf(MAXLINE), c
   integer i, lastc
   data lastc /MAXLINE/, buf(MAXLINE) /NEWLINE/
   # megjegyzés: MAXLINE = MAXCARD + 1

   lastc = lastc + 1
   if (lastc > MAXLINE) {
      read(STDIN, 100, end = 10) (buf(i), i = 1, MAXCARD)
         100 format(MAXCARD a1)
      lastc = 1
      }
   c = buf(lastc)
   getc = c
   return

10 c = EOF
   getc = EOF
   return
   end
```

A **getc** beolvas egy teljes sort, majd ezt karakterenként adja tovább. Az új sorokat szükség szerint olvassa, és minden sor végén az utolsó aktuális kártyaoszlop után elhelyez egy **NEWLINE** karaktert. A **MAXCARD** és a **MAXLINE** a kártya hosszát (valószínűleg 80), ill. a sor hosszát jelölő szimbolikus konstansok; különbségük 1.

A **read** utasítás az **end=** szerkezetet használja a file végének érzékelésére, és így a **getc** képes az **EOF** visszaadására. Amikor a file végére érünk, a vezérlés a megjelölt utasításnak adódik át. Ez a lehetőség széleskörűen használatos, de egyes rendszerek esetleg más formát is igényelhetnek. A **STDIN** azt a logikai perifériaszámot jelöli, amelyet a rendszer a FORTRAN inputra használ. Ez sok esetben az 5-ös logikai perifériaszám.

A **putc** nyilvánvalóan a **getc** inverze; egyenként gyűjti a karaktereket egy sorba, és ezt a sort — ha szükséges, szóközökkel kitöltve — kiírja, amikor egy **NEWLINE**-nal mint aktuális paraméterrel hívjuk meg. **STDOUT** a FORTRAN output logikai perifériaszáma, amely sok esetben a 6-os logikai szám.

```
# putc (egyszerű változat) — karakterkivitel a standard outputra
   subroutine putc(c)
   character buf(MAXCARD), c
   integer i, lastc
   data lastc /0/
   if (lastc > MAXCARD | c == NEWLINE) {
      for (i = lastc + 1; i <= MAXCARD; i = i + 1)
         buf(i) = BLANK
      write(STDOUT, 100) (buf(i), i = 1, MAXCARD)
         100 format(MAXCARD a1)
      lastc = 0
      }
```

```
if (c ¬= NEWLINE) {
     lastc = lastc + 1
     buf(lastc) = c
     }
return
end
```

Van néhány problematikus részlet a **putc**-ban. Mi történjen pl. akkor, ha egy teljes sornál hosszabb szöveg érkezik **NEWLINE** nélkül? A legkézenfekvőbb megoldásnak az tűnik, hogy a sort mindenképpen kiírjuk, ha a puffer megtelt — amint az a mi változatunkban is történik. A hosszú sorok figyelmeztetés nélküli csonkítása kevésbé kívánatos.

A rögzített hosszúságú, szóközökkel kitöltött rekordábrázolásnak egy másik gyenge pontja az, hogy a sor végén ténylegesen létező szóközöket nem lehet megkülönböztetni a kitöltésre használt szóközöktől. Olyan környezetben, amely a szóközzel való kitöltést erőlteti, a legcélszerűbb, ha a **getc** eltávolítja a befejező szóközöket, a **putc** pedig újra előállítja őket. Bár ezáltal némi információ elvész, azon alkalmazások száma, ahol ez problémát okoz, nem jelentős.

Azt is feltételeztük, hogy amikor egy sort a **putc**-vel kiírunk (vagyis amikor egy **NEWLINE**-nal hívjuk meg), a sor ténylegesen megjelenik, ha a program a hívást követően egy **stop** utasítással befejezi működését. Ha a rendszerünk nem nyújtja ezt a szolgáltatást, akkor a program befejezésekor a **putc**-n belül kell a maradék kiviteléről gondoskodni. Egyik lehetőség a **putc**-nek olyan módosítása, hogy a

```
call putc (EOF)
```

kikényszeríti az összes, addig felgyülemlett karakter kivitelét.

A könyvben szereplő kisebb programok zöme végrehajtási idejének túlnyomó részét a karakterek be- és kivitele teszi ki, még akkor is, ha ezek hatékonyan vannak megvalósítva. Minden bizonnyal bőségesen megtérül az az erőfeszítés, amit ezen egyszerű változatok tökéletesítésére fordítunk.

Gyakorlatok

1.12 Módosítsuk a **getc** és a **putc** eljárást oly módon, hogy eltávolítsa, ill. visszaállítsa a sorvégi szóközöket! □

1.13 Dolgozzunk ki egy módszert a sorhatároknak rögzített hosszúságú kártyákon való jelzésére, és módosítsuk a **getc**, valamint a **putc** utasítást úgy, hogy ezt alkalmazza! Ez módot ad arra, hogy a változó hosszúságú rekordokból származó előnyöket kihasználjuk akkor is, ha egy adott rendszer ezeket nem engedi meg. □

1.10 HOGYAN TOVÁBB?

Mit végeztünk eddig? Megírtunk több elemi, de hasznos eszközt. Megpróbáltuk úgy felépíteni őket, hogy könnyen érthetőek és szükség esetén egyszerűen módosíthatók legyenek. Olyan nyelvet használtunk, amelyet változtatás nélkül le lehet fordítani és futtatni jóformán minden számítógépen. A programok megírása során olyan elemi műveleteket használtunk, amelyek elfedik az operációs rendszerek közti különbségeket.

A következőkben ugyanezt a folyamatot akarjuk alkalmazni több olyan eszközre, amelyek szerintünk hasznosak lesznek az olvasó számára, és amelyekből különböző, a programozással kapcsolatos tanulságokat vonhatunk le. Ezen programok legtöbbje nagyobb lesz, mint az eddigiek, némelyikük sokkal nagyobb. Ennek ellenére, a gondos strukturálás és a programnak az operációs rendszertől való elkülönítése fő témánk marad. Ez az egyetlen módja annak, hogy megbirkózzunk a nagy programokkal és a valódi rendszerekkel.

Vegyük majd észre, hogy a programok gyakran használnak az előző szakaszokban vagy fejezetekben leírt programrészeket. Ez azért lényeges, mert a nagyobb szoftver- produktivitás elérésének egyik módja, hogy arra építünk, ami már megvan, ahelyett, hogy állandóan újra meg újra feltalálnánk ugyanazokat a dolgokat kis változtatásokkal. Egy könyvben azonban ez a szervezési mód terhet ró az olvasóra, mivel egy adott program futtatásához szükséges részek nincsenek mind egy helyen. Megpróbáltunk ezen könnyíteni oly módon, hogy gondosan választott neveket, emlékeztetőket, visszautalásokat és részletes tárgymutatót alkalmaztunk. Ennek ellenére néha nehéz lesz egy fejezet közepén elmélyülni az anyagban és azonnal felfogni, hogyan működnek a programok.

A jó programozás elsajátításának egyik legjobb módszere a tényleges programokról való olvasás és gondolkodás, és olyan kérdések föltevése, hogy „miért csinálták ezt ilyen módon" vagy „miért nem csinálták ezt másképp?" Hiszünk abban, hogy a könyvben található programok az olvasó számára tőkét jelentenek, nem pedig kötelezettséget, és reméljük, hogy hasznára lesz tanulmányozásuk még akkor is, ha ő másképp írta volna ezeket.

BIBLIOGRÁFIAI MEGJEGYZÉSEK

Az egész könyvben feltételezzük, hogy az olvasónak gyakorlati ismeretei vannak a FORTRAN-ról. Ha ez kiegészítésre szorul, ajánlhatjuk D. D. McCracken könyvét: *A Simplified Guide to FORTRAN Programming* (Wiley, 1974). Ez olyan bevezetést ad, amely elkerüli a fölösleges matematizálást, és a nyelv kedvező lehetőségeinek bemutatására törekszik.

A FORTRAN ellenőrző program, amelyet használtunk, B. G. Ryder: „The PFORT verifier" című cikkében található (*Software—Practice and Experience,* 1974. okt.). Ez a cikk meglehetősen részletesen tárgyalja a portábilis FORTRAN-részhalmazt. A B. W. Kernighan: „RATFOR — a preprocessor for a rational FORTRAN" (*Software — Practice and Experience,* 1975. okt.) a RATFOR nyelv további leírását adja, különös tekintettel a tervezésével kapcsolatos megfontolásokra. A RATFOR-on kívül sok más FORTRAN előfeldolgozó létezik, a 9. fejezet végén található bibliográfia utal néhány lehetőségre.

2. fejezet

Szűrők

Folytatjuk, amit az előző fejezetben elkezdtünk — olyan egyszerű programok írását, amelyek egy standard inputról olvasnak és egy standard outputra írnak. Ezeket a programokat az elektronikával (vagy a vízvezeték-szereléssel) való nyilvánvaló analógia alapján szűrőknek nevezzük, mivel egy átmenő adatfolyamon hajtanak végre hasznos változtatásokat. Látni fogjuk, hogy nagyon sok eszköz, beleértve a könyvben találhatók legtöbbjét, ebbe a kategóriába tartozik.

2.1 TABULÁTOROK VISSZAÁLLÍTÁSA

Kezdjük az **entab** szűrőnek, a **detab** inverzének elkészítésével. Az **entab** a szóközökből álló sorozatokat tabulátorkarakterekkel és szóközökkel helyettesíti. Emlékeztetünk arra, amit korábban az abból származó előnyökről mondtunk, hogy file-jaink a lehetőségekhez képest legjobban hasonlítanak egymáshoz. Az **entab** szűrőket arra használhatjuk, hogy kártyákat olvasunk be és írógépszerű szöveget állítunk elő. Ilyen módon file-jainkat olyan alakba írhatjuk, amelyben nincsenek fölösleges belső szóközökből álló jelsorozatok. Egy további előny, hogy file-jaink kisebb helyfoglalásúak és azonos alakúak; ez megkönnyíti olyan programok írását, amelyek jól együttműködnek.

Az **entab** egy másik felhasználása olyan kimenet előállítása, amely írógép jellegű terminálra küldhető. Lehet pl. olyan programunk, amely egy sornyomtatót működtet. Úgy szeretnénk a kiírást felgyorsítani, hogy ahol csak lehet, tabulátorokat használunk. Egy működő program átírása helyett jobban járunk egy külön programmal — az **entab**-bal —, amely a kiírás előtt megszűri a kimenő adatot.

A legtöbb szűrőt akkor terveztük meg jól, ha megtaláltuk a bemenő adatok felismerésének olyan módszerét, amely lehetővé teszi, hogy az adatok sorrendjét a program vezérlési struktúrája tükrözze, nem pedig kapcsolók és flag-ek tömege. A **wordcount**-ban, amely egy kis program, elegendő volt egyetlen flag, de nagyobb programokat flag-ek erdeje igen zavarossá tehet. Ha úgy tekintjük az **entab** adatait, mint a következő minta ismétlődését: nulla vagy több szóköz, amelyet egy nemszóköz jel (vagy **EOF**) követ, akkor ez meghatározza a program vezérlési szerkezetét:

```
col = 1
repeat {
    while (getc(c) == BLANK)       # szóközök gyűjtése
        if (tabulátorpozíció)
            egy tab kivitele
    if (maradt még szóköz)
        vigyük ki őket
```

```
    # c most EOF vagy nemszóköz
    if (c == EOF)
        break
    call putc(c)
    if (c == NEWLINE)
        col = 1
    else
        col = col + 1
    }
```

A **col** a kimenő karakter aktuális pozíciója.

A **break** utasítás — amit eddig még nem használtunk — azonnali kilépést eredményez az őt tartalmazó ciklusból, ami **while, for, repeat** vagy **do** lehet. Egy **repeat** ciklus, amelyből hiányzik a befejező **until** bejegyzés, végtelen ciklus, azaz olyan, amelyben nincs ellenőrzés, csupán egy feltétel nélküli ugrás a végéről az elejére. Ilyen ciklus végrehajtását valami más módon kell befejezni; jelen esetben a **break** utasítást használjuk, amelyet egy **EOF** megjelenésekor hajtunk végre. Ciklusokat akkor írunk ilyen formában, amikor egy vagy több utasítást kell végrehajtani mind a ciklus ismétléseit vezérlő vizsgálat előtt, mind pedig utána.

A **break** utasítást *nem szabad* fegyelmezetlenül használnunk a megfelelően átgondolt **while** vagy **until** vizsgálat helyett! Különösen ügyeljünk az olyan ciklusokra, amelyekből több helyről lehet kilépni. Amikor egy ciklus különböző részeiből több **break** segítségével lépünk ki, nagyon nehéz az összes lehetséges kilépés egységes viselkedését biztosítani. Mindig meg kell próbálnunk a kilépési feltételt — származzon az egy **while** vagy **until** ellenőrzési részéből, vagy legyen az egy **break** utasítást vezérlő **if** — egyetlen logikai kifejezésként megfogalmazni.

A szóközök számontartásának egyszerű módja az **entab**-ban egy újabb változó, a **newcol** használata, amely eltávolodik a **col**-tól, ha szóközök következnek. Amikor egy tabulátorkarakter kerül kivitelre, a **col**-t egyenlővé tesszük a **newcol**-lal. Ezután, amikor egy nemszóköz és nem-tabulátorkarakter érkezik, akkor — ha a **col** kisebb, mint a **newcol** —, ez többletszóközök összegyűlését jelenti (de nem akkora számban, hogy tabulátorral helyettesíthetők lennének). Ezeket a szóközöket ki kell vinnünk a karakter elé. A feladat megoldása:

```
# entab — szóközök helyettesítése tab-okkal és szóközökkel
    character getc
    character c
    integer tabpos
    integer col, i, newcol, tabs(MAXLINE)

    call settab(tabs)
    col = 1
    repeat {
        newcol = col
        while (getc(c) == BLANK) {     # szóközök gyűjtése
            newcol = newcol + 1
            if (tabpos(newcol, tabs) == YES) {
                call putc(TAB)
                col = newcol
                }
```

```
        }
    for (; col < newcol; col = col + 1)
        call putc(BLANK)            # maradék szóközök kivitele
    if (c == EOF)
        break
    call putc(c)
    if (c == NEWLINE)
        col = 1
    else
        col = col + 1
    }
stop
end
```

A **tabpos** és a **settab** természetesen ugyanazok a rutinok, amelyeket a **detab** használt az első fejezetben; a programot megint úgy szerveztük, hogy a tabulátorpozíciók belső ábrázolását elfedi a főprogram elől. Figyeljük meg a **for** utasítás kezdőérték-beállítás nélküli használatát! Ebben az esetben az előző programrész elvégez minden szükséges kezdőértékadást.

Gyakorlatok

2.1 Vizsgáljuk meg az **entab** működését olyan file-on, amelyekben kettő, egy, ill. nulla számú karakter van! Próbáljuk ki olyan sorokkal, amelyek 0-tól 10-ig terjedő számú szóközt tartalmaznak, amelyet egy **x** követ! Próbáljuk ki 0-tól 10-ig terjedő számú szóközzel, amelyet a „file vége" követ! Milyen körülmények között áll vissza egy file eredeti formájába azután, hogy a **detab**-bal és az **entab**-bal egymás után feldolgoztuk? Találnánk-e alkalmazást ehhez a művelethez? □

2.2 Mi történik, ha **entab** egy tabulátorkaraktert olvas? Készítsük el a legegyszerűbb kiegészítést a programhoz, amely a tabulátorkaraktereket helyesen kezeli! Hogyan írnánk át a programot, hogy az szóközökből és tabulátorokból álló tetszőleges sorozatokat ugyanilyen karakterekből álló, azonos jelentésű és minimális hosszúságú sorozatokra képezzen le? Mit csinál az **entab** olyan szöveggel, amelyik visszalépéseket tartalmaz? □

2.2 PROGRAMOK PARAMÉTEREZÉSE

Az **entab** ugyanazt a konvenciót használja, mint a **detab:** minden nyolcadik oszlop tabulátorpozíció. Jó lenne azonban, ha volna egy egyszerű módszerünk, amellyel ezeknek a programoknak paraméterként egy tabulátorállítási listát tudnánk átadni, amely ideiglenesen megváltoztatná a szokásos beállítást. Néhány operációs rendszer lehetőséget ad arra, hogy hozzáférjünk a program parancssorához vagy vezérlőkártyájához, amely a programot behívta, és ily módon esetleges paramétereket vagy más információt adjunk át. Ha a rendszerünk ilyen, akkor gondoskodnunk kell a **getarg** elemi függvényről, amely a paraméterinformációt elérhetővé teszi a program számára. Bemutatott programjaink többsége előnyösen tud majd használni néhány (nem kötelező) paramétert. Néhány program esetében a paraméterek megadása kötelező lesz. A továbbiakban feltételezzük, hogy a **getarg** valamilyen módon megvalósítható.

A **getarg**-ot a következőképp terveztük. A

getarg(n, array, maxsize)

eljárás átmásolja az *n*-edik paraméter karaktereit az **array** egész típusú tömbbe, egy-egy karaktert téve minden egyes tömbelembe. Mint korábban, most is egészeket használunk karakterek tárolására. A **maxsize** mutatja azt a maximális karakterszámot, amelynek kezelésére fel vagyunk készülve; a **getarg** elhagyja a paraméter végét, ha ez szükséges ahhoz, hogy az beférjen a fenntartott helyre. A **getarg** a paraméter hosszát (a karakterek számát) adja vissza saját értékeként, ha az *n*-edik paraméter létezik; ha *n*-nél kevesebb paraméter van, akkor **EOF**-et ad vissza. Ez azt sugallja, hogy az **EOF** negatív érték legyen, hogy könnyen megkülönböztethessük egy érvényes hossztól. Megfontolhatjuk azt is, hogy az **EOF** nagy szám legyen, nagyobb, mint az elképzelhető **maxsize** értéke, de ez kevésbé jó megoldás. Azonban bárhogyan is választunk, a programunknak minden esetben össze kell hasonlítania a **getarg**-tól visszakapott értéket az **EOF**-fel és soha nem szabad kihasználnia valamilyen, az **EOF** értékével kapcsolatos rejtett információt. Ha ezt az elvet figyelmen kívül hagyjuk, ki leszünk téve annak, hogy a programunk rejtélyes módon működésképtelenné válik, ha a környezete megváltozik. Minél jobban függ egy program az őt befogadó rendszer sajátosságaitól, annál kevésbé rugalmas.

A **getarg** által az **array**-ben elhelyezett paraméter ezután karaktersorozatként használható. Nehézséget okoz azonban az, hogy bár tudjuk a tömbben levő karakterek számát, ez az információ nem része a tömbnek. Sokkal kényelmesebb, ha egy karaktertömb hossza magában a tömbben van elhelyezve.

Két lehetőségre gondolhatunk. Az egyik az, hogy a tömb első eleme, **array(1)** tartalmazhatná az őt követő karakterek számát. A másik lehetőség, hogy a tömb *végét* valamilyen speciális értékkel jelöljük meg, olyannal, amelyik nem azonos egyetlen valódi karakterrel sem. Mindkét megoldásnak vannak előnyei és hátrányai (ennek átgondolását az olvasóra bízzuk). Némi tépelődés után a sorozat végének jelzése mellett döntöttünk. Programjainkban minden karaktersorozat utolsó elemként egy speciális jelzőt tartalmaz, melyet **EOS**-nek nevezünk. Egy egyetlen karakterből álló karaktersorozat esetén **array(1)** tartalmazza a karaktert és **array(2)** az **EOS**-t; üres karaktersorozat esetén **array(1)** tartalmazza az **EOS**-t.

Ily módon az utolsó dolog, amit **getarg**-nak el kell végeznie, annak megszervezése, hogy az általa visszaadott karaktersorozat **EOS**-sel legyen lezárva, hossza pedig a **maxsize** által szabott határon belül legyen.

Ha lehetőségünk van arra, hogy a paraméterlista formátumát saját magunk tervezzük meg, akkor az látszik legkényelmesebbnek, ha a **getarg** feltételezi, hogy a paramétereket egy vagy több szóköz és/vagy tabulátor választja el. (Ha olyan paramétereket is kezelnünk kell, amelyen belül szóközök vagy tabulátorkarakterek lehetnek, akkor megengedhetjük, hogy a paraméterek nem kötelezően kiírandó idéző jelek közé legyenek zárva, melyeket a **getarg** eltávolít.) Úgy gondoljuk, hogy programok számára való paraméterátadáskor az elválasztóként szereplő vesszők és zárójelek csak zavart és többlet írásmunkát okoznak, ezért jobb ezeket elkerülni.

Gyakorlatok

2.3 Írjuk meg az **echo** nevű programot, amely a paramétereket a standard outputra másolja. A szomszédos paramétereket egy szóköz választja el, az utolsó után pedig új sor következik. Eszünkbe jut-e az **echo**-nak a **getarg** tesztelésén kívül más alkalmazása is? □

2.4 Módosítsuk a **detab**-ot és **entab**-ot oly módon, hogy paraméterként egy tabulátorpozíciókból álló listát vegyen át, és így a felhasználók a programot olyan utasításokkal hívhatják, mint

detab 9 17 25 33 41
entab 10 16 33 73

Természetesen a paramétereket értelmező és a **tabs** tömböt feltöltő programban óvatosan kell eljárni, nehogy túltöltsük a tömböt. Mindkét programnak ésszerűen kell dolgoznia, ha nincsenek paraméterek. (Ugyanebben a fejezetben később előfordul a **ctoi** program, amely karaktersorozatokat egészekké konvertál.) □

2.5 Terjesszük ki úgy a programokat, hogy elfogadják a következő rövidített írásmódot:

entab $m + n$

melynek jelentése: tabulátorpozíció az m-edik oszloptól számított minden n-edik oszlopban. □

2.3 ÁTÜTÉSEK

Egy írógépen úgy tudunk jeleket átütni, hogy visszalépünk a már leírt szövegre. Ennek segítségével húzunk alá szavakat, és ez a módszer arra is használható, hogy újabb jeleket állítsunk elő a már létezők segítségével. Azonban, ha az outputot egy sornyomtatón írjuk ki, az eredmény értelmetlen lehet, mivel a sornyomtató általában nem tud mit kezdeni a visszaléptető karakterekkel.

Ugyanakkor sok sornyomtató képes teljes sorok átütésére. Ennek vezérlésére a FORTRAN egy külön vezérlő karaktert vezet be: a sor elejére kiküldött szóköz azt jelenti: „emelj sort nyomtatás előtt”, a pluszjel (+) pedig azt, hogy „ne emelj sort nyomtatás előtt”, azaz üsd át a korábbi sort.

Az **overstrike** nevű szűrő megkeresi a visszaléptető karaktereket a gépelt szövegben, és vezérlő kódokkal ellátott sornyomtató sorok sorozatát állítja elő, amely a visszaléptető karakterek hatását helyettesíti. Az eredményként kapott program nagyon hasonló lesz az **entab**-hoz, ha megtartjuk az ott elfogadotthoz hasonló szemléletünket, amely szerint a bemeneti jelsorozat nulla vagy több visszaléptetett karakterből és egyéb karakterekből áll. Ha az **overstrike** egy vagy több visszaléptető karakterből álló sorozatot talál, akkor befejezi az aktuális sort és elhelyezi a megfelelő számú szóközt az átütő sorban.

Természetesen ez nem az egyetlen módja a megoldásnak, de ez az egyik legegyszerűbb. A program kellemetlenül fog viselkedni, ha a kinyomtatandó szöveg betűnként aláhúzott szavakat tartalmaz. Ebben az esetben ugyanis minden *karakter, visszalépés, aláhúzássorozat* egy új teljes sor előállítását váltja ki, ami meglehetősen lassú. Ezért jobb módszer lehet két vagy több sorpuffer használata, amelyben az átütési kombinációk szükség szerint építhetők fel. Ezt azonban nehezebb programozni és belőni. Sokszor hasznos megelégedni valamivel, ami a feladatok legnagyobb részét jól elvégzi, a tökéletesítésekre és a bővítésekre pedig akkor térni át, ha már tudjuk róluk, hogy megérik a fáradságot.

Az **overstrike** jó szolgálatot tesz olyan szövegek esetén is, amelyek nem tartal-

maznak visszalépést, mivel az írógépes szöveget — amit a jelen könyvben levő legtöbb program is előállít — vezérlő karakterekkel ellátott sorokká alakítja, melyek sornyomtatón kiírathatók. Végső szűrőként használható, valahányszor sornyomtatón kihozott eredményre van szükség, és megint csak arra ösztönöz minket, hogy amennyire csak lehetséges, tartsuk magunkat a más programok számára készített szövegek standard belső formájához.

```
# overstrike — visszalépéseket tartalmazó sorok helyettesítése több-
    # szörös sorokkal
  character getc
  character c
  integer max
  integer col, newcol

  col = 1
  repeat {
      newcol = col
      while (getc(c) == BACKSPACE) # visszalépések kiküszöbölése
          newcol = max(newcol — 1, 1)
      if (newcol < col) {              # indul az átütő sor
          call putc(NEWLINE)
          call putc(NOSKIP)
          for (col = 1; col < newcol; col = col + 1)
              call putc(BLANK)
          }
      else if (col == 1 & c ¬= EOF)    # normál sor kezdete
          call putc(SKIP)
                                       # különben sor belseje
      if (c == EOF)
          break
      call putc(c)                     # normál karakter
      if (c == NEWLINE)
          col = 1
      else
          col = col + 1
      }
  stop
  end
```

A **max** függvény írását az olvasóra hagyjuk, ennek feladata paraméterei közül a legnagyobbnak a megtalálása. PL/1-ben a **max** beépített függvény. FORTRAN-ban a **max0** használható, amely két vagy több egész számnak a maximumát választja ki, és egyetlen egész értéket ad vissza. Az **&** a logikai és, ami a FORTRAN-ban az **.and.**.

A programban a **NOSKIP** és a **SKIP** szimbolikus konstansokat használtuk a **PLUS** és **BLANK** helyett, mivel ezek jobban leírják a végrehajtandó tevékenységet, és az utóbbiak nem is általánosak. Emellett lényeges, hogy különbséget tegyünk a soremelést előidéző szóköz és egy sor kitöltésére használatos szóköz között. A szimbolikus konstansok célja az, hogy megtartsuk a mnemonikus információt, ameddig csak lehetséges.

Ebben az esetben az **else if** sorozatban nincs szükségünk egy utolsó **else**-re,

mivel a harmadik alternatíva előfordulásakor (a soron belül) nem kell semmit tennünk. Ennek ellenére elhelyeztünk egy megjegyzést a rejtett alternatíva megvilágítására. Gyakran fogunk **else if** láncokat magyarázatként kitett **else**-zel befejezni, hogy az összes lehetőséget felsorolhassuk.

Gyakorlatok

2.6 Melyik az a féltucat próbaadat, amely kipróbálja az **overstrike** kritikus határereseit? Miért írtuk azt, hogy

newcol = max(newcol — 1, 1)

ahelyett, hogy

newcol = newcol — 1

Adjunk próbaadatot és példaeredményeket, amelyek a két eset közötti különbséget kimutatják. A mi **overstrike** változatunk olyan terminál viselkedését szimulálja, ahol a visszalépések figyelmen kívül maradnak, ha a bal lapszélt elértük. Módosítsuk a programot úgy, hogy megfelelően számon tartsa a karakterek megjelenésének helyét, de csak azokat a karaktereket nyomtassa ki, amelyek az első oszlopban vagy ettől jobbra fordulnak elő! □

2.7 Mint korábban már említettük, ha a visszalépések egyenként fordulnak elő és nem sorozatokban egymás után (például, ha egy szóban a betűk egyenként vannak aláhúzva), az **overstrike** nem hatékony, hiszen minden visszalépésre egy új sort visz ki. Módosítsuk úgy, hogy kevesebb sort hozzon létre! □

2.8 Egy másik szabványos FORTRAN vezérlőkarakter az **1**, amelyik a következő sort új oldal tetejére nyomtatja. Módosítsuk az **overstrike** szűrőt úgy, hogy figyelje egy speciális **FORMFEED** karakter előfordulását, és ennek hatására egy lapváltást állítson elő. Gondolhatunk az **overstrike** olyan módosítására is, amely figyeli az üres sorok hosszabb sorozatait, és amikor lehetséges, lapváltással helyettesítse ezeket, mivel ez a sornyomtatókon általában gyorsabb. □

2.9 Az **overstrike** a **detab**-bal együtt egy általános írógép—sornyomtató leképezést valósít meg. Érdemes lenne-e egyetlen programban egyesíteni őket? Tudnánk-e más olyan tevékenységet mondani, amellyel érdemes ezeket kiegészíteni? □

2.4 SZÖVEGTÖMÖRÍTÉS

A tabulátorok és a visszalépések bizonyos értelemben rövidítésekként foghatók fel; mindenesetre az írógépkódú file-ok általában rövidebbek, mint a kártyaképek vagy a sornyomtató sorok. De ez a rövidítésnek nagyon speciális esete. A következőkben egy ismétlődő, karaktersorozatokat tartalmazó szövegek tömörítésére és kifejtésére képes rendszert vizsgálunk meg. Az ismétlődő karakterek nem szükségképpen szóközök, hanem bármilyen más jelek is lehetnek.

Hangsúlyozzuk, hogy ez nem a végső tömörítési lehetőség. Egy file-ban jelentős mennyiségű redundáns információ fordulhat elő, amely nem szomszédos karakterek ismétléseként jelenik meg, gondoljunk például egy szótárra. De a kártyaképek és a sornyomtatott sorok hajlamosak arra, hogy hosszú szóközsorozatokat tartalmazza-

nak, és a számítógép outputja általában gyakran tartalmaz ismétléseket. Ezért számíthatunk arra, hogy a mi viszonylag egyszerű megoldásunk sok helyen hasznosnak fog bizonyulni.

Való igaz, hogy egy file, amelyben nincsenek ismétlődések, nem lesz rövidebb a „tömörítés" után, sőt, valójában egy kicsit hosszabb lesz. Minden tömörítési mód alapja az, hogy van valamelyes ismeretünk a bemenő információ szerkezetéről (különben nem volna redundáns információ, amit kihagyhatna). Ezért minden módszerhez található olyan speciális eset, amelyik a tömörítéstől csak rosszabb lesz. Egy olyan file, amely véletlen információt tartalmaz, jó tesztanyag, mivel várhatóan nem tömöríthető.

Egy tömörítési módszer teherbírásának egyik ismérve, hogy mennyire rossz eredményt ad a számára kedvezőtlen esetekben. Nem akarjuk, hogy egy nem megfelelő file teljes „információrobbanást" idézzen elő, de elviseljük, ha az eredmény egy kicsit hosszabb lesz; ez elkerülhetetlen és nem katasztrofális. Tartsuk ezeket az elveket szem előtt a program felépítése során. Később még visszatérünk rájuk.

Az általunk használt módszer abban áll, hogy folytonosan figyeljük az azonos karakterek ismétlődéséből álló sorozatokat. Ha ilyet találunk, eredményül valami olyasmit akarunk kivinni, ami ezt jelenti: „itt van egy ilyen és ilyen karakterekből álló sorozat, ami ilyen és ilyen hosszú". A sorozatok között kivisszük a maradék szöveget oly módon, hogy azt megelőzi valami, ami a következőt jelenti: „Itt jön egy maradék szöveg, ami ilyen és ilyen hosszú". A nagy maradékokat vagy a hosszú sorozatokat kisebb darabok kezelhető sorozataként lehet kivinni. A kimenő információ a következő szerkezetek váltakozásából áll:

az ismétlés kódja
az ismétlendő karakter
ismétlésszámláló

és

maradékszámláló
megfelelő számú karakter

Mint mindig, most is az a fontos, hogy olyan szemléletet alkossunk a bemenő adatokról, ami megkönnyíti a program körvonalainak kialakítását. Ebben az esetben a bemenetet egy vagy több azonos karakter ismétlődéséből álló sorozatok egymásutánjának tekintjük. Minden sorozat hosszát összehasonlítjuk a **THRESH** küszöbértékkel, amely egyelőre tetszőleges. Ha egy sorozat hossza eléri vagy meghaladja a **THRESH**-t, akkor azt kódolt sorozatként visszük ki; különben egy belső pufferba tesszük, ahonnan egy maradék részeként kerül később kivitelre. Első közelítésünk a következő:

```
for (lastc = getc(lastc); lastc ¬= EOF; lastc = c) {
    for (nrep = 1; getc(c) == lastc; nrep = nrep + 1)
                                        # sorozat gyűjtése
        ;
    if (nrep < THRESH)
        for ( ; nrep > 0; nrep = nrep — 1)     # rövid sorozat
            lastc átvitele a pufferbe
    else {
        pufferbeli maradék kivitele
```

```
            call putc(RCODE)
            call putc(lastc)
            ismétlésszámláló kivitele
            }
        }
    pufferbeli maradék kivitele
```

Itt a **for** utasítás felhasználásának két, eddig nem látott változatát találhatjuk. A külső **for** azt biztosítja, hogy valahányszor csak a ciklus törzse lefut, a **lastc** mindig az utolsó, fel nem dolgozott karaktert tartalmazza. Nem történik semmilyen számlálás, ez mégis épp annyira megengedett módja egy ciklus vezérlésének, mint bármi más. Ez a **for**-nak a hagyományos **do**-énál nagyobb fokú rugalmasságát bizonyítja.

Az első belső **for**-nak nincs más törzse, mint egy pontosvessző. A feladatot teljes egészében ellátja az ellenőrző rész, ahol a **getc** minden híváskor egy újabb karaktert ad, valamint a léptetési rész, amely megnöveli **nrep**-et. A pontosvessző a **for** végét jelzi, különben az az őt követő **if**-et próbálná vezérelni. Az ily módon használt pontosvesszőt *üres utasításnak* nevezzük. Legjobb a pontosvesszőt egyedül hagyni egy sorban, mivel különben könnyen szem elől téveszthető és a kód félreérthető. Az üres utasítás mindenütt használható ugyan, ahol egy RATFOR utasítás megengedett, általában mégis csak üres ciklustörzsként szokták alkalmazni; a többi előfordulásait célszerű a program átnézésekor eltávolítani.

Még ki kell találnunk, hogyan tegyünk karaktereket a pufferba és hogyan vigyük ki azokat. Mivel a kivitel két különböző helyen történik, célszerű szubrutint hívni a feladatra. Egy karakternek a pufferba való átvitele nemcsak egy indexnövelést és egy indexelt értékadást igényel, hanem a puffer betelésének az index értéke alapján való vizsgálatát is magában kell hogy foglalja. Ha ez bekövetkezik, előáll a harmadik eset, amikor a puffert üríteni kell. A szubrutinra tehát határozottan szükség van.

Mekkora legyen a puffer? Nincs okunk rá, hogy nagyobbra vegyük, mint amekkora a kivihető legnagyobb maradék, amelyet viszont az egy karakterként ábrázolható számok intervalluma korlátoz. Ez az intervallum az egy kódolással kifejezhető ismétlődések számát is korlátozza, mivel a számlálót egyetlen karakterben kell tárolnunk.

Ez felveti a második, még alapvetőbb szempontot: hogyan ábrázoljunk számlálókat karakterekként? Eddig elkerültük, hogy sokat mondjunk a karakterek belső- vagy külső ábrázolásáról. Ameddig minden karakter kódja különböző, és az egyenlőség eldöntése érdekében egész számokat hasonlítunk össze, bármilyen módszer megteszi. Valóban, e könyvben az általunk megírt formában szereplő majdnem mindegyik program független a karakterek belső reprezentációjától.

A **compress** kivétel. (Más kivételek az **expand**, amely a **compress** inverze és a **crypt** rejtjelező program. Ezeket a következő két pontban tárgyaljuk.) A **compress** esetében kifejezhetnénk a számlálókat numerikus karakterek sorozatával, és így továbbra is elkerülhetnénk a problémát, de ez pazarlás, és mi éppen szövegtömörítéssel foglalkozunk.

A legjobb megoldás, ha a **getc** az összes karaktert pozitív számok hiánytalan sorozatára képezi le, amely 0-val vagy 1-gyel kezdődik. (Néhány FORTRAN változatban az **r1** input formátum elintézi ezt, de ez nem szabványos.) Az **EOF**-hez és az **EOS**-hez hasonló kódoknak kis negatív értékeket adhatunk, a fejezet elején már tárgyalt okokból. A **putc** az inverz leképzést végzi el kis pozitív egész számokról külsőleg ábrázolható karakterekre.

Ilyen körülmények között a számlálók olvasásakor és írásakor semmi különleges átalakításra nincs szükség. Mindössze annyit kell tennünk, hogy egy nagy karakterértéket foglalunk le az **RCODE** ismétlési kódnak, és a maximális számlálóértéket ennél kisebbre vesszük. Egy másik lehetőség, hogy az **RCODE**-ra a zérus kódot használjuk, mivel amúgy sem akarunk 0 számláló érték mellett kivitelt végrehajtani. Egy nyolcbites karakterábrázolással rendelkező számítógépen például a **getc** a karaktereket belsőleg a 0-tól 255-ig terjedő egész számokkal ábrázolhatja. Ha **RCODE**-ot 255-nek vesszük, akkor a **MAXCHUNK** 254 lehet. Ha **RCODE** értéke zérus, a maradékok 255 karakter hosszúak lehetnek.

Célszerű, ha az olvasó maga győződik meg arról, hogy ez a módszer még akkor is hibátlanul működik, ha az **RCODE** maga is előfordul a tömörítendő szövegben.

```
# compress — szövegtömörítés a standard inputról
   character getc
   character buf(MAXCHUNK), c, lastc
   integer nrep, nsave

   # kell, hogy RCODE > MAXCHUNK vagy RCODE = 0 legyen
   nsave = 0
   for (lastc = getc(lastc); lastc ¬= EOF; lastc = c) {
        for (nrep = 1; getc(c) == lastc; nrep = nrep + 1)
             if (nrep >= MAXCHUNK)      # ismétlődés számlálása
                  break
        if (nrep < THRESH)              # rövid sorozat
             for ( ; nrep > 0; nrep = nrep — 1) {
                  nsave = nsave + 1
                  buf(nsave) = lastc
                  if (nsave >= MAXCHUNK)
                       call putbuf(buf, nsave)
                  }
        else {
             call putbuf(buf, nsave)
             call putc(RCODE)
             call putc(lastc)
             call putc(nrep)
             }
        }
   call putbuf(buf, nsave)             # utolsó maradék kiiratása
   stop
   end

# putbuf — buf(1) ... buf(nsave) kivitele, nsave törlése
   subroutine putbuf(buf, nsave)
   character buf(MAXCHUNK)
   integer i, nsave
```

```
    if (nsave > 0) {
        call putc(nsave)
        for (i = 1; i <= nsave; i = i + 1)
            call putc(buf(i))
    }
    nsave = 0
    return
    end
```

Egy stiláris megjegyzés: amikor egy rövid karaktersorozatot csatlakoztattunk a pufferhoz, megvizsgáltuk **nsave**-t, hogy eléri vagy meghaladja-e maximális biztonsági értékét. (Ugyanezt tettük **nrep**-pel is.) Továbbá, a **putbuf** végén **nsave**-t zérusra állítottuk, még akkor is, ha az **nsave**>0 feltétel nem teljesült. Ennek a tevékenységnek ugyanis csak akkor van jelentősége, ha **nsave** negatív. Ugyanakkor a program olvasásából kitűnik, hogy **nsave** sosem haladhatja meg **MAXCHUNK**-ot és sosem lehet negatív, miért törődjünk hát vele?

Ez az, amit „defenzív programozás"-nak neveznek. Szerepeltetése jóformán semmibe sem kerül sem a forrásszövegben, sem a végrehajtási időben, mégis csökkenti annak az esélyét, hogy a program megvaduljon, ha egy fontos vezérlő változó valahogy megsérül. Nem lehet *mindenhol* hibaüzeneteket nyomtatni, de lehet és kell biztosítani magunkat, ahol erre mód van.

Sokkal könnyebb belőni azt a programot, melynek a kimenete nem terjedelmes és amelynél egy hiba mellékhatásaként nem lép fel még a tár felülírása is. Az ésszerűbb viselkedést az általunk javasolt módon lehet biztosítani. Az **if, while** és **until**-ban szereplő vizsgálatokat úgy adjuk meg, hogy a lehetetlen helyzeteket a biztonságos irányba tereljék. Használjuk arra **else—if** láncok utosó **else**-ét, hogy a „soha" be nem következő feltételeket, amelyek azonban mégis előadódhatnak, kezeljék biztonságosan. Soha ne vizsgáljunk valamit egyenlőségre, ha ugyanazt a „nagyobb vagy egyenlő", ill. a „kisebb vagy egyenlő" esetre is vizsgálhatjuk. Közelebbről, ne hagyjunk ciklusokat ismétlődni, amikor egy változó kívül esik a számára megengedett intervallumon. Ne írjunk olyan programot, amely „vonzza" a hibákat!

Miért írtuk azt, hogy

```
for (nrep = 1; getc(c) == lastc; nrep = nrep + 1)
    if (nrep >= MAXCHUNK)
        break
```

ahelyett, hogy

```
for (nrep = 1; getc(c) == lastc & nrep < MAXCHUNK;
    nrep = nrep + 1)
```

Ez ravasz kérdés, amelynek semmi köze nincs az üres utasításokhoz. A FORTRAN-ban, mint a jelenleg használt legtöbb nyelvben, nincs arra mód, hogy megjelöljük az **&**-tel és **|**-el összekapcsolt logikai kifejezések kiértékelésének sorrendjét. A fordítóprogram a vizsgálatokat tetszőleges sorrendben végezheti, és kihagyhat belőlük, ha a logikai értékre nélkülük is következtetni tud. A legtöbb esetben ez nem jelent különbséget. Itt azonban a **getc** hívásának meg kell történnie, különben **c** nem kapja meg helyes értékét. Ha először **nrep**-et vizsgáljuk és a **getc** nem hívódik meg, az

eredmény egy nagyon alattomos hiba. Ezért írtunk két külön utasítást, bár látszólag egy is elégnek tűnt volna.

Egy ciklus vagy egy **if** vezérlésére mindig egyetlen logikai kifejezés használatát részesítjük előnyben, mert így a kapott program jobban olvasható. De amikor a kiértékelés sorrendje fontos, olyan programot írunk, amelyik jól működik anélkül, hogy bármilyen fordítóprogram által hozott döntésektől függene. Ily módon a prog ram futtatható lesz egy másik fordítóprogrammal és nem keletkeznek benne rejtélyes hibák.

Most térjünk vissza korábbi, a tömörítési módszerekkel kapcsolatos tárgyalásunkhoz, hogy eldönthessük, mire állítsuk be a **THRESH** küszöbértéket? Ha **THRESH** nem nagyobb 1-nél, akkor minden karakter ismétlődő jelsorozattá válhat. Az **abcdef** jelsorozat a következőképpen kódolódik:

RCODE
a
1 *(ismétlésszámláló)*
RCODE
b
1
RCODE
c
1
stb.

Az output háromszorosa az inputnak, ami nem egészséges dolog. (A számok számlálók, nem pedig karakterek.)

A kettes küszöbérték már sokkal jobban kezelné ezt az esetet. Minden maradék szövegnek minősül, így **MAXCHUNK** számú karakterenként csak egy kód kerülne kivitelre. Ugyanakkor az **abbabba** file a következőképpen kódolódna:

1 *(maradékszámláló)*
a
RCODE
b
2 *(ismétlésszámláló)*
1 *(maradékszámláló)*
a
stb.

Ez minden három inputjelre öt outputjelet jelent, ami még mindig nem jó. Ha a küszöb három, a módszer még mindig veszteséges lehet.

Négyes küszöbértékre még rosszul viselkedő karaktersorozatok esetén sem fizetünk rá, ötnél pedig mindig nyerünk az ismétlések kódolásával. Ennél nagyobb értékek már nem ösztönzik az ismétlések kódolását, még ha az helymegtakarítással járna is, ezért ezek már nem célszerűek. Természetesen soha nem tudjuk kiküszöbölni azt a pesszimális határesetet, amikor nincsenek ismétlődések, és ilyenkor egy többletkarakter jön létre minden **MAXCHUNK** számú karakter után; ez azonban nem ér minket váratlanul. A fentiekből következik, hogy a legjobb a **compress**-t a **THRESH** értékének ötre állítása mellett futtatni.

Gyakorlatok

2.10 Írjuk le pontosan, hogy milyen körülmények között, milyen hibát tartalmazó eredményeket kapunk, ha a **compress**-ben a fentiek alapján a

```
for (nrep = 1; getc(c) == lastc & nrep < MAXCHUNK;
    nrep=nrep + 1)
```

utasítást használjuk. □

2.11 Hogyan tudnánk kihasználni egy szótárban (rendezett szavak listája, definíciók nélkül) levő redundanciát abból a célból, hogy azt minimális terjedelműre kódoljuk át? Próbáljuk ki a módszerünket oly módon, hogy a kedvenc szótárunk 73. oldalán levő szavakat átkódoljuk! □

2.5 SZÖVEGKIFEJTÉS

Most, hogy már van egy módszerünk szövegek tömörítésére, szükségünk van egy társprogramra is, amelyik újra kifejti a szövegeket, hogy azok más programokban felhasználhatók legyenek. Tudjuk, hogy a bemenet ismétlődésekből és tömörítetlen maradékokból áll, melyeket egy kódkarakter előz meg, és ez megmondja, hogy a két maradék közül melyik következik és az hány karakter hosszúságú. Ezért első közelítésként a következőt írhatjuk:

```
while (getc(code) ¬= EOF)
    if (code == RCODE) {      # ismétlődés
        c = getc(c)
        for (code = getc(code); code > 0; code = code — 1)
            call putc(c)
        }
    else
        for ( ; code > 0; code = code — 1)      # maradék
            call putc(getc(c))
```

Hibátlan bemenet esetén ez kitűnően dolgozik, de mi történik, ha egy **EOF** érkezik, amikor **c**-be olvasunk? A program erre nem készült fel, és így megpróbál egy vagy több **EOF** kódot átadni **putc**-nek. Esetleg ismételten hívja **getc**-t. Korábban nem határoztuk meg, hogy ezek az elemi műveletek hogyan működnek ilyen esetben. Bár elvileg ésszerűen kellene viselkedniük, mégis lényeges annak biztosítása, hogy az őket hívó program az inputtól függetlenül is ésszerűen viselkedjék.

Megoldásként közbe tudnánk iktatni egy másik függvényt, amelyet a **getc** helyett hívnánk, és amely emlékezne az **EOF**-re és kizárná a további hívásokat. Ezzel együtt járna egy olyan függvény, amelyet a **putc** helyett kell hívni, és amely eltünteti az **EOF**-eket. A **getc**-t és **putc**-t közvetlenül is módosíthatnánk. Bármelyik választás biztosítaná a program helyes működését, ha egy file végén túl olvasunk be vagy írunk ki. Ez lehetővé tenné a kevésbé gondos programozást. Ilyen egyszerű megoldás azonban nem mindig adódik, ezért inkább bátran szembenézünk a hibaellenőrzés alapvető kérdéseivel.

Azt is megtehetnénk, hogy az új beviteli program megáll, amikor **EOF**-et talál, hiszen nincsen semmi befejező tevékenység, amit el kellene végeznie. Néha ez a legjobb vagy talán az egyetlen dolog, amit tenni lehet. Mi azonban megpróbáljuk elke-

rülni az ilyen megoldásokat, bármilyen kényelmesnek is tűnnek első pillanatra, ugyanis megsértik a felülről lefelé tervezés alapelvét: minden függvénynek vissza kell térnie oda, ahonnan hívták. Ily módon a stratégia látható (és módosítható) marad a program legmagasabb szintjén, a végrehajtás pedig szigorúan felülről lefelé történik.

Ezért az **expand** működő változata **getc** minden hívása után megvizsgálja, hogy **EOF** jött-e vagy sem.

```
# expand — szövegkifejtés a standard inputról
    character getc
    character c, code

    while (getc(code) ¬= EOF)
        if (code == RCODE) {       # ismétlődés kifejtése
            if (getc(c) == EOF)
                break
            if (getc(code) == EOF)
                break
            for ( ; code > 0; code = code — 1)
                call putc(c)
            }
        else {                     # maradék kifejtése
            for ( ; code > 0; code = code — 1) {
                if (getc(c) == EOF)
                    break
                call putc(c)
                }
            if (c == EOF)
                break
            }
    stop
    end
```

A jó kiértékelési sorrend biztosítása érdekében újra több utasításban kell felírnunk egy logikai kifejezést:

```
if (getc(c) == EOF)
    break
if (getc(code) == EOF)
    break
```

Figyeljük meg, hogy az **else** részben két **break**-re van szükség, hogy kijussunk egy kétszeresen skatulyázott ciklusból, a belső **for**-ból és a külső **while**-ból.

A hibaellenőrzések, azáltal, hogy elfedik a program viselkedését alapvetően befolyásoló események folyamatát, kétségkívül rontják annak olvashatóságát, mégis szükség van rájuk. A fő problémát az okozza, hogy végrehajtásuk érdekében kénytelenek vagyunk a programra egy olyan struktúrát ráerőltetni, amely nem következik az elvégzendő feladat lényegéből. Azok a programok azonban, amelyek kezdettől fogva jól átgondolt hibaellenőrzést tartalmaznak, megbízhatóbbak és hosszabb életűek, mint azok, ahol a hibaellenőrzést utólagos meggondolások folyamán ragasztottuk a programokhoz.

2.12 Határozzuk meg, hogy mit kell a **getc**-nek és **putc**-nek tennie **EOF** esetén az őket hívó programok egyszerűsítése érdekében. Írjuk át az **expand**-ot, hogy ki tudja használni ezt a módosítást! □

2.13 Mi történik, ha az **expand** első változata egy tetszőleges (tömörítetlen) file-t kap? Ha ugyanezt a végső változat kapja? Mi történik, ha egy karakter eltűnik egy hibátlan tömörített file-ból? El tudunk-e képzelni valamilyen módszert az olvasás újraszinkronizálására ezzel a kódolási rendszerrel? Tudnánk-e mutatni olyan kódolási rendszert, amelyben az újraszinkronizálás könnyebb? Mi ezen kódolási rendszer legrosszabb esete a kifejtésben, és hogyan viselkedik véletlen adatokat tartalmazó file-lal kapcsolatban?

2.14 Az **RCODE** csak egybites információt tartalmaz, amely azt mondja meg, hogy a következő tétel ismétlődés-e vagy nem. Ha a számláló teljes intervallumát ritkán használjuk ki, előnyös lehet megfelezni a megengedhető intervallumot és az **RCODE**-ot ugyanabban a karakterben tárolni, mint ahol a számlálót. Adjunk erre egy áttekinthető módszert és írjuk át ennek megvalósításaként a **compress**-t és az **expand**-ot. Szükséges-e, hogy a számlálók intervalluma 2 hatványa legyen? És az, hogy 2 többszöröse legyen? Milyen **THRESH** érték lenne célszerű? □

2.15 Bizonyítsuk be, hogy minden megfordítható tömörítési módszer, amely bármilyen inputot elfogad és bizonyos file-ok méretét csökkenti, az bizonyos file-ok méretét szükségképpen növeli. □

2.6. REJTJELEZÉS

A számítógépes file-okkal kapcsolatos titoktartás biztosításának egyik módja a rejtjelezés, amikor az eredeti adatokat olyan ábrázolásban tároljuk, amely nem érthető meg a rejtjelezés inverze, a megfejtés nélkül. Igaz, hogy a legtöbb számítógéprendszer nyújt valamilyen fokú védelmet az ellen, hogy mások elolvassák file-jainkat, mégis majdnem mindig vannak ennek a védelemnek gyenge pontjai, melyek kívül esnek hatáskörünkön. Az itt szereplő **crypt** program használatával saját bizalmas információink védelmét valósíthatjuk meg.

Módszerünkben a **key** (kulcs) karaktersorozatot használjuk, melyet mi adhatunk meg, és amely meghatározza, hogy hogyan kell megváltoztatni a standard input-ról jövő minden egyes karaktert, mielőtt átvinnénk a standard outputra. A kulcs első karakterét használjuk az első bemenő karakterrel, a másodikat a másodikkal, és így tovább. Ha a kulcs rövidebb, mint az input — ami rendszerint így van — a kulcsot újra elölről kezdjük, ahányszor csak szükséges.

```
# crypt — rejtjelezés — rejtjelfejtés
   character getc, xor
   character c, key(MAXKEY)
   integer getarg, mod
   integer i, keylen
```

```
        keylen = getarg(1, key, MAXKEY)
        if (keylen == EOF)
            call error("hívás: crypt key.")
        for (i = 1; getc(c) ¬=EOF; i=mod(i, keylen) + 1)
            call putc(xor(c, key(i)))
        stop
        end
```

Az **error** olyan elemi művelet, amely egy adott üzenetet nyomtat ki (az üzenet végét pont jelzi, ami már nem kerül kiírásra) és megáll. Igaz, hogy ezzel megsértettük azt az elvet, amit korábban kifejtettünk a hibakezelésről (legalábbis akkor, ha a főrutinon kívülről hívjuk az **error**-t), de vannak olyan helyzetek, amelyek túl zavarosak ahhoz, hogy a feldolgozást folytathassuk. Bizonyos esetekben semmilyen magas szintű stratégia sem biztosíthatja a program folytatását, legalábbis anélkül nem, hogy teljesen össze ne zavarná a program alapvető szándékát. Az **error**-t végzetes hibák szabványos jelzésére fogjuk használni.

Hogyan használjuk a kulcs egyes karaktereit? Nagyon sok olyan függvény létezik, amely két karakterből előállít egy harmadik, szabványos kimenő karaktert, de binárisan kódolt karakterek esetén a legmegfelelőbb függvény a *kizáró vagy*. Ez olyan neveken is ismert, mint *szimmetrikus összeg, átvitel nélküli összeadás,* vagy *tagadott ekvivalencia*. A kizáró vagyra nincsen szabványos szimbólum, mi a ⊕-t használjuk.

A Boole-algebra segítségével a kizáró vagy a következőképpen definiálható:

a ⊕ b = (a & ¬b) | (¬a & b)

Az eredmény igaz, ha **a** és **b** logikai értéke különböző, és hamis, hogyha azonos. Két karakteren elvégzett kizáró vagy művelet azt jelenti, hogy a karakterek belső reprezentációját alkotó, egymásnak megfelelő biteken hajtjuk végre a műveletet. A szabványos FORTRAN-ban nincsenek bitekre vonatkozó logikai műveletek, de az **and**, **or** és **not** függvények sok konkrét megvalósításban fordulnak elő. Ha az **xor** kizáró vagy függvény közvetlenül nem használható, más logikai függvények segítségével is kifejezhetjük:

```
    # xor — a kizáró vagy függvény
        character function xor(a, b)
        character and, not, or
        character a, b

        xor = or(and(a, not(b)), and(not(a), b))
        return
        end
```

A **compress**-hez és az **expand**-hoz hasonlóan ez a függvény is megköveteli, hogy a szabványos karakterek belső ábrázolása egy kitöltött intervallumba essen. Egy további megszorítás, hogy a karakterek ábrázolása valamilyen *n* mellett az *n* bites számok teljes halmazát tegye ki. Ez annyit jelent, hogy a karakterek száma 2-hatvány kell hogy legyen. Különben előfordulhatna, hogy a kulcs egy karakterének és egy bemenő karakternek az **xor**-ral képzett eredménye olyan kódot ad, amelyet nem tudunk kivinni. Ilyen esetben valamilyen más, kevésbé jó függvényt leszünk kénytelenek használni.

A kizáró vagy előnye a szimmetria:

a ⊕ b ⊕ a = b
a ⊕ b ⊕ b = a

A kiértékelés sorrendje közömbös. Ez azt jelenti, hogy egy file-t ugyanazzal a programmal (és természetesen ugyanazzal a kulccsal) fejthetünk meg, amellyel rejtjeleztük, hiszen a kulcs kétszeri alkalmazása visszaadja azt, amivel indultunk. Nem kell tehát kitalálnunk azt, hogy hogyan semlegesítsük az egyik program hatását, és ebből a célból egy másik programot írnunk.

Ez azt is jelenti, hogy egy file-t kétszeresen rejtjelezhetünk, majd utána könnyedén megfejthetjük. Tegyük fel például, hogy először a **brillig** kulccsal rejtjeleztünk egy file-t. A rejtjelezett változatot még egyszer rejtjelezzük a **frumious** kulcsot használva. Az információt eredeti formájára hozhatjuk a két kulcs *tetszőleges sorrendben* való újra alkalmazásával, mivel a kizáró vagy szimmetrikus. A végeredmény az, mintha *egyszer* rejtjeleztünk volna egy olyan kulccsal, melynek hossza a két kulcs hosszainak legkisebb közös többszöröse, ami jelen esetben $7 \times 8 = 56$ karakter. A kombinált kulcs már olyan karaktersorozat, amely nem hasonlít semmilyen beszélt nyelvű (pl. angol) szövegre, ezért nehezebb kitalálni. A rejtjelezett kód megfejtése számottevően megnehezült. Egyébként hangsúlyozzuk, hogy ennél sokkal hatásosabb rejtjelezési módszerek is vannak, és ez a módszer az egyik legrégibb és legismertebb. Hacsak a kulcs nem nagyon hosszú, a CIA vagy a NSA várhatóan egy napon belül megfejti file-unkat. És még hogyha nem is kerülünk kapcsolatba ezekkel a szervezetekkel, a legtöbb számítóközpont szórakozásból is foglalkozik rejtvényfejtéssel.

Alkalmi próbálkozások vagy véletlen betekintések ellen azonban a **crypt** megfelelő biztosítékot szolgáltat. Ha valaki megfejti file-jainkat, az valószínűleg azért történt, mert a kulcsot hozzáférhető helyen felejtettük! Amíg egyszerű módszerekkel célt tudunk érni, ne foglalkozzunk bonyolultabbakkal.

Gyakorlatok

2.16 Mit kezdünk egy rejtjelezett file-lal, aminek a kulcsát elfelejtettük? □

2.17 Ha ismerjük egy file első egy-két sorának eredeti változatát, valamint rendelkezésünkre áll a rejtjelezett változat, hogyan lehetne meghatározni a rejtjelezésnél használt kulcsot? Milyen eszközöket használnánk erre? □

2.18 Írjuk meg az **xor (a,b)** függvényt nemnegatív **a** és **b**-re úgy, hogy csak szabványos FORTRAN műveleteket használunk, azaz nem használunk **and**-et **or**-t és **not**-ot. Milyen gyors ez más, hozzáférhető rutinokkal összehasonlítva? Mennyire érzékeny a gép szóhosszára és a gépi aritmetika típusára? □

2.7. KARAKTEREK ÁTÍRÁSA

A szűrők egyik osztálya a rajta átmenő karakterek egy részét átírja, a többit pedig módosítás nélkül továbbengedi. Olyan programot szeretnénk — a **translit**-et —, amely a

translit x y

hívásra a standard inputról jövő minden **x**-et **y**-nal helyettesíti a standard outputon. A többszörös átírás szintén hasznos:

```
translit xy yx
```

minden **x**-et **y**-ra cserél és minden **y**-t **x**-re. Továbbá jó lenne, ha használhatnánk a következő rövidítést:

```
translit a—z A—Z
```

ami az összes kisbetűt nagybetűre változtatja. A

```
translit a—zA—Z A—Za—z
```

hívás a nagybetűket kicsikre, a kicsiket pedig nagyokra fordítja át. Még a jó gépírók is előnyben részesítik a

```
translit A—Z a—z
```

írásmódot, a

```
translit ABCDEFGHIJKLMNOPQRSTUVWXYZ abcdefghijklmnop
     qrstuvwxyz
```

írásmóddal szemben.

Ha a rövidítéseket kiküszöböltük és a paramétereket már kifejtettük, az átírási ciklus egyszerű:

```
while (getc(c) ¬= EOF) {
     i = index(from, c)
     if (i > 0)     # sikeres megfeleltetés
          call putc(to(i))
     else     # sikertelen megfeleltetés
          call putc(c)
     }
```

A **from** tartalmazza a lefordítandó karakterek halmazát, a megfelelő fordítások pedig **to**-ban találhatók. Az **index** eljárás megadja a **c** karakter **from**-beli indexét, vagy nullát ad, ha **c** nem található **from**-ban. A PL/1-ben az **index** egy beépített függvény (karaktersorozatokra, de nem a mi kibővített kódjainkra); RATFOR-ban a következőképpen írható meg:

```
# index — c keresése az str sorozatban
   integer function index(str, c)
   character c, str(ARB)

   for (index = 1; str(index) ¬= EOS; index = index + 1)
        if (str(index) == c)
             return
   index = 0
   return
   end
```

Az **ARB** szimbolikus konstansot egy olyan tömbméret-deklarációban használjuk, melynek értéke tetszőleges, mivel a tömböt paraméterként adjuk át, és mérete közömbös a rutin számára. **ARB** általában nagy, pozitív érték.

Vannak olyan esetek, amikor egy teljes karakterosztályt egyetlen karakterre szeretnénk lefordítani és ezután a lefordított karakter ismétlődéséből álló sorozatokat egyetlen példánnyá tömöríteni. Például a szóközöket, tabulátorokat és újsor-karaktereket újsor-karakterekre fordítjuk, és utána ezek ismétlődéseit egyetlen újsor-karakterrel helyettesítjük. Eredményül olyan szöveget kapunk, amelyben minden szó külön sorban foglal helyet, további feldolgozásra készen. Vagy azt is akarhatnánk, hogy egy program szövegében minden alfabetikus szimbólum *a*-kra fordítódjék és az összes szám *n*-ekre. Az ilyen tömörítéses műveletet oly módon jelölhetjük, hogy a második paraméter rövidebb az elsőnél:

```
translit a—zA—Z a
translit 0—9 n
```

Ez azt eredményezi, hogy a második paraméter utolsó karakterét (a **to** karaktersorozatot) annyiszor kell ismételni, hogy a két paraméter hossza egyenlő legyen, és ez az ismételt karakter soha nem jelenhet meg kétszer közvetlenül egymás mellett a kimenő jelsorozatban.

A **translit** főrutinja a következő lesz:

```
lastto = length(to)
if (length(from) > lastto)
    collap = YES
else
    collap = NO
repeat {
    i = index(from, getc(c))
    if (collap == YES & i >= lastto) {        # tömörítés
        call putc(to(lastto))
        repeat
            i = index(from, getc(c))
            until (i < lastto)
        }
    if (c == EOF)
        break
    if (i > 0)        # fordítás
        call putc(to(i))
    else              # másolás
        call putc(c)
    }
```

A **length** egy rövid rutin, amelyik egy jelsorozat hosszát határozza meg (az **EOS**-t nem beleszámítva). Azért használjuk, hogy be tudjuk állítani **collap** értékét, amely azt mutatja, hogy kell-e tömöríteni vagy sem:

```
# length — karaktersorozat hossza
    integer function length(str)
    integer str(ARB)

    for (length = 0; str(length + 1) ¬= EOS; length = length + 1)
        ;
    return
    end
```

A **translit** lehetőséget ad arra, hogy egyes karaktereket lefordítsunk, míg másokat tömörítünk, de ezt nem azért terveztük így, mintha arra számítanánk, hogy ez gyakran előfordul, hanem azért, hogy a program a paraméterektől függetlenül észszerű és előrelátható módon viselkedjék.

A tömörítés csak azokat a karaktereket érinti, amelyek pozíciójuk szerint az utolsó fordítási karakternek felelnek meg (vagy ezután állnak). A **lastto** index a **to** tömb utolsó karakterére mutat, hogyha ez a tömb rövidebb, mint a **from**, egyébként a **to**-ban levő **EOS**-re, és nem érinti a fordítást.

Van azonban egy hiba: ha a **to** hiányzik vagy üres, és csak egyetlen **EOS**-t tartalmaz, minden fordítás **to(0)**-ra fog hivatkozni. (Lehet, hogy az olvasó erről maga is meg akar győződni.) Adhatnánk egy hibaüzenetet, amikor a **translit** második paraméter üres, de ezt az esetet olyan utasításnak is tekinthetjük, amely a **from**-beli összes karaktert töröltei, valahányszor csak előfordulnak az input szövegben. Ez egy hasznos és kézenfekvő értelmezés. A megvalósításához szükséges változtatások kézenfekvőek, ha szem előtt tartjuk, hogy a **lastto==0** mindig a kiválasztott karakter törlését írja elő. Kiválasztott karakter az, amelyre az **index** nullánál nagyobb.

Még egy utolsó lehetőség megteremtése célszerű: Néha szeretnénk átírni és tömöríteni egy karakterhalmaz kivételével valamely szöveg összes karakterét: Például a

translit ¬ a—z —

az összes, nem kisbetűből álló karaktersorozatot egyetlen kötőjellel helyettesíti. Az első paraméter előtt álló ¬ jel „nem”-ként olvasandó, akárcsak a RATFOR forrásprogramokban. A

translit ¬ a—z

a kisbetűk kivételével minden karaktert töröl.

A kiegészítés újra egyszerű, mivel a program funkcionális modulokra oszlik. Csupán két dolgot kell bevezetnünk: az **allbut** jelzőt, melyet **YES**-re állítunk, amikor egy adott halmaztól eltekintve az összes karaktert át kell írni, valamint egy új függvényt, a **xindex**-et, amely az **index**, és a program többi része között tart kapcsolatot.

```
# xindex — az index által adott állapot inverze
    integer function xindex(array, c, allbut, lastto)
    character array(ARB), c
    integer index
    integer allbut, lastto
```

```
if (c == EOF)
    xindex = 0
else if (allbut == NO)
    xindex = index(array, c)
else if (index(array, c) > 0)
    xindex = 0
else
    xindex = lastto + 1
return
end
```

Ha **allbut** tartalma **NO**, **xindex** az **index** által adott értéket szolgáltatja. Amikor **allbut** tartalma **YES**, **xindex** „hazudik": ha **index** megtalálja a karaktert, **xindex** letagadja ezt, ha pedig **index** nem találja meg, **xindex** azt mondja, hogy megvan. Továbbá feltesszük, hogy a néhány megadott karakteren kívül a halmaz olyan nagy, hogy csak egyetlen karakterre való leképezésüknek (vagy törlésüknek) van értelme, így ha **allbut** tartalma **YES**, **xindex** olyan **indexet** ad vissza, amely tömörítésre utal, ha a karakter benne van a halmazban; zérust ad vissza, ha nincs benne, végül az **index** eredményét adja, ha **allbut** értéke **NO**. A **xindex** arra is vigyáz, hogy **EOF**-et nem tekinthet kiválasztott karakternek.

Ilyen szervezés mellett **index** egyszerű marad, és ugyancsak nem lesz bonyolult az a program, amely őt **xindex**-en keresztül használja. Képzeljük el, milyen lenne a program logikája, ha a fentinek megfelelő döntések elszórtan szerepelnének a főprogramban, ahelyett, hogy a **xindex**-ben vannak koncentrálva!

Mellesleg megjegyezzük, hogy az **xindex**-et a következőképpen is írhattuk volna:

```
xindex = index(array, c)
if (c == EOF | (allbut == YES & xindex > 0))
    xindex = 0
else if (allbut == YES & xindex == 0)
    xindex = lastto + 1
```

Ez rövidebb, de kevésbé áttekinthető. Olyan logikai döntések, amelyek együtt használják az & és | operátorokat vagy zárójeleket igényelnek, nehezen érthetők, ezért kerülendők. Az általuk használt változat tisztán mutatja, hogy négy eset közül pontosan egyet kell kiválasztani. Ha kétségeink vannak, próbáljuk ki a „telefon teszt"-et: ha megértünk egy logikai kifejezést, amikor hangosan felolvassák, akkor az eléggé világos. Különben át kell írni.

Most már csupán az maradt hátra, hogy elkészítsük a paraméter-értelmező programot, és egy hatékony karakterfordító birtokába jutottunk:

```
# translit — karakterek leképezése
character getc
character arg(MAXARR), c, from(MAXSET), to(MAXSET)
integer getarg, length, makset, xindex
integer allbut, collap, i, lastto
```

```
if (getarg(1, arg, MAXARR) == EOF)
    call error("hívás: translit from to")
else if (arg(1) == NOT) {
    allbut = YES
    if (makset(arg, 2, from, MAXSET) == NO)
        call error("from: túl nagy")
    }
else {
    allbut = NO
    if (makset(arg, 1, from, MAXSET) == NO)
        call error("from: túl nagy")
    }
if (getarg(2, arg, MAXARR) == EOF)
    to(1) = EOS
else if (makset(arg, 1, to, MAXSET) == NO)
        call error("to: túl nagy")
lastto = length(to)
if (length(from) > lastto | allbut == YES)
    collap = YES
else
    collap = NO
repeat {
    i = xindex(from, getc(c), allbut, lastto)
    if (collap == YES & i >= lastto & lastto > 0) {   # tömörítés
        call putc(to(lastto))
        repeat
            i = xindex(from, getc(c), allbut, lastto)
            until (i < lastto)
        }
    if (c == EOF)
        break
    if (i > 0 & lastto > 0)       # fordítás
        call putc(to(i))
    else if (i == 0)              # másolás
        call putc(c)
    }                             # különben törlés
stop
end
```

Figyeljük meg, hogy a hiányzó paraméterek esetén kinyomtatott üzenet nemcsak azt mondja, hogy „nincs paraméter", hanem a figyelmeztető szöveg emlékezteti a felhasználót arra, hogyan kell a programot helyesen használni. Jobb tömören megmondani, hogy mi a helyes eljárás, mint csupán a hibára hívni fel a figyelmet.

Ettől eltekintve a **translit**-ot úgy írtuk meg, hogy kevés hibaüzenetet ad. A szokatlan paramétereknek amikor csak lehet, valamilyen ésszerű értelmezést adunk, különben pedig „ártalmatlanul" dolgozzuk fel őket. Ez általános célú eszközök számára jó tervezési elv. Ha nem így járunk el, akaratlanul is kizárhatunk olyan hasznos alkalmazásokat, melyekre induláskor nem gondoltunk. A fenti elv csökkenti a hibaüzenetek áradata által okozott zavart is.

A **makset** létrehozza a **from** és **to** halmazokat a **filset** és **addset** segítségével.

```
# makset — halmaz előállítása array(k)-ból set-ben
    integer function makset(array, k, set, size)
    integer addset
    integer i, j, k, size
    character array(ARB), set(size)

    i = k
    j = 1
    call filset(EOS, array, i, set, j, size)
    makset = addset(EOS, set, j, size)
    return
    end
```

Az **addset** egy tömb megjelölt pozíciójára helyez el egy karaktert és megnöveli az indexet. Azt is ellenőrzi, hogy van-e hely ennek elvégzésére. Ezt a függvényt gyakran alkalmazzuk majd a későbbi programokban.

```
# addset — c beírása set(j)-be, ha belefér
    integer function addset(c, set, j, maxsiz)
    integer j, maxsiz
    character c, set(maxsiz)

    if (j > maxsiz)
        addset = NO
    else {
        set(j) = c
        j = j + 1
        addset = YES
        }
    return
    end
```

A **translit** egy rövidítést vezet be egymás után következő kisbetűk, nagybetűk és számjegyek részére, anélkül, hogy bármilyen feltételezéssel élne ezen karaktereknek az egész változóban való ábrázolásáról. A **filset** végzi el a fordítási halmaz felépítésének teljes munkáját, szükség szerint kifejtve a rövidítéseket az **addset**, **dodash**, **esc** és az **index** segítségével. A **filset**-et általános formában írtuk meg, mivel arra számítunk, hogy majd később is használjuk karakterhalmazokkal való manipulációkra. A rutin egy tetszőleges elhatárolót figyel, és a módosított indexértékeket adja vissza.

```
# filset — array(i)-nél kezdődő fordítási halmaz kifejtése delim-ig
    subroutine filset(delim, array, i, set, j, maxset)
    character esc
    integer addset, index
    integer i, j, junk, maxset
    character array(ARB), delim, set(maxset)
    string digits "0123456789"
    string lowalf "abcdefghijklmnopqrstuvwxyz"
    string upalf "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

```
for ( ; array(i) ¬= delim & array(i) ¬= EOS; i = i + 1)
    if (array(i) == ESCAPE)
        junk = addset(esc(array, i), set, j, maxset)
    else if (array(i) ¬= DASH)
        junk = addset(array(i), set, j, maxset)
    else if (j <= 1 | array(i + 1) == EOS)    # — konstans
        junk = addset(DASH, set, j, maxset)
    else if (index(digits, set(j — 1)) > 0)
        call dodash(digits, array, i, set, j, maxset)
    else if (index(lowalf, set(j — 1)) >0 )
        call dodash(lowalf, array, i, set, j, maxset)
    else if (index(upalf, set(j — 1)) > 0)
        call dodash(upalf, array, i, set, j, maxset)
    else
        junk = addset(DASH, set, j, maxset)
return
end
```

A **string** deklaráció az a rövidítés, amelyet a programoknak a hosszú kezdőérték-beállításoktól való mentesítése érdekében használunk. FORTRAN-ban a

```
string id "xyz"
```

a következőre fejtendő ki:

```
integer id(4)
data id(1) /LETX/
data id(2) /LETY/
data id(3) /LETZ/
data id(4) /EOS/
```

ahol a **LETX** stb. szimbolikus konstansok a megfelelő betűk belső ábrázolására. A 8. fejezet tárgyalja, hogy a **string**-et hogyan lehet automatikusan lefordítani egy makroprocesszor segítségével.

A **filset**-ben levő másik új dolog a

```
junk = addset(...)
```

alakú hívások halmaza. Mivel **filset**-et nem érdekli az **addset** által visszaadott érték (a végső ellenőrzést a **makset** végzi), azt gondolhatnánk, hogy ettől az értéktől megszabadulhatunk, ha a

```
call addset(...)
```

alakú hívást használjuk. Bár ez sokkal esztétikusabb, sajnos egyes FORTRAN-változatok valóban különbséget tesznek szubrutinok és függvények között, és zavarja őket, ha egy függvényt **call**-lal hívunk. Ha azt akarjuk, hogy a programjaink valóban portábilisak legyenek, meg kell védenünk magunkat ettől oly módon, hogy a függvényeket mindig *függvényként* hívjuk. A felesleges függvényértékeket mindig a **junk**-ba fogjuk beírni, utalva arra, hogy ezáltal tulajdonképpen „eldobjuk” őket.*

* junk jelentése: hulladék *(A lektor).*

A **translit** egy úgynevezett *felhívójel* (escape)-*konvenciót* is tartalmaz a tabulátorok és az újsor-karakterek olyan módon való leírására, hogy azok láthatók legyenek, és a lehető legkevesebb nehézséget okozzák annak a programnak, amely a paramétereket vizsgálja. Felhívójelként az @ (egységár) jelet használjuk, amelyik karakter a felhívójel után áll, az valamilyen módon speciális. Közelebbről, @**t**-t tabulátorként, @**n**-et pedig újsorként definiáljuk. Ily módon írhatjuk a következőt:

```
translit " @t @n" @n
```

ami az összes „fehér helyeket" (szóközöket, tabulátorokat és újsor-karaktereket) egyetlen újsor-karakterré változtatja és ezáltal soronként egy-egy szót helyez el. (Figyeljük meg az idézőjelek használatát, amelyek, mint már korábban említettük, lehetőséget adnak arra, hogy az első paraméterben egy szóközt helyezzünk el.) Ha szükséges, más speciális kódok is könnyen létrehozhatók. A felhívójel megszünteti az ő követő karakterek speciális jelentését, beleértve a szóközöket, idézőjeleket és magát a felhívójelet is, ezért a speciális karaktereket közvetlenül használhatjuk. A fenti példát így is írhattuk volna:

```
translit @  @t@n @n
```

feltéve, hogy **getarg** szintén ismeri a felhívójel-konvenciót.

A felhívójel világos és egységes módszert ad speciális karakterek jelentésének megváltoztatására. Az **esc** rutin végzi el a felhívójel ellenőrzését, valamint a megfelelő karakter és index létrehozását:

```
# esc — array(i) cseréje speciális karakterre, ha szükséges
  character function esc(array, i)
  character array(ARB)
  integer i

  if (array(i) ¬= ESCAPE)
       esc = array(i)
  else if (array(i + 1) == EOS)      # @ utolsóként nem speciális
       esc = ESCAPE
  else {
       i = i + 1
       if (array(i) == LETN)
            esc = NEWLINE
       else if (array(i) == LETT)
            esc = TAB
       else
            esc = array(i)
       }
  return
  end
```

Az **esc** is a **xindex**-hez hasonlóan a bonyodalmakat egy egyszerű kapcsolattartó rutinban rejti el ahelyett, hogy a döntéseket szétterítené a programban.

A **filset** legnagyobb része egy többszörös elágazás. Hogyha a vizsgálat alatt levő karakter nem kötőjel, vagy előtte egy felhívójel áll, akkor közvetlenül hozzárakjuk a

halmazhoz. Ugyanezt tesszük az olyan kötőjelekkel is, amelyek elsőként vagy utolsóként fordulnak elő, vagy nem betű, illetve számjegy előzi meg őket. Csupán a betűk vagy számjegyek között megjelenő kötőjel jelenthet rövidítést, és csak ez kerül kifejtésre. Ezt a **dodash** végzi:

```
# dodash — kötőjeles paraméter kifejtése
   subroutine dodash(valid, array, i, set, j, maxset)
   character esc
   integer addset, index
   integer i, j, junk, k, limit, maxset
   character array(ARB), set(maxset), valid(ARB)

   i = i + 1
   j = j — 1
   limit = index(valid, esc(array, i))
   for (k = index(valid, set(j)); k <= limit; k = k + 1)
        junk = addset(valid(k), set, j, maxset)
   return
   end
```

A **translit** megértés szempontjából nem könnyű program, mivel igen sokféle dolgot csinál, ugyanakkor tevékenységeit érdemes volt kombinálni, mivel mindegyik hasonló problémára vonatkozik. Megtanuljuk a leghasznosabb alkalmazásokat, mint pl. a

```
translit A—Z a—z
```

amely nagybetűről kisbetűre való átalakítást végez, és a többi formai szabályt csak akkor kell szemügyre vennünk, ha újabb alkalmazásra kerül sor. A program maga azonban nem bonyolult, mivel modulárisan építettük fel, a legegyszerűbb alkalmazásokkal kezdve. A későbbiekben hozzátett bonyolultabb részek különálló új modulokra oszlanak és nem szabdalják szét az eredeti szerkezetet.

Mi határozza meg a **translit** futási idejét? Amint ez egy olyan programnál várható, amelyik kevés számítást végez, a **translit** a központi egységre vonatkozó idejének legnagyobb részét B/K (I/O, azaz bemeneti—kimeneti) célú feldolgozással tölti. Esetünkben ez 70—85%. Gépidejének legnagyobb részét azonban *nem* a **getc** és **putc** teszi ki, hanem az ezek által hívott alacsonyabb szintű rutinok, melyekre nincsen befolyásunk. Karaktereknek a **getc**-vel és **putc**-vel való egyenkénti kezelése elhanyagolható időt igényel. A B/K után a központi egységben töltött idő maradéka az **xindex**-ben és az **index**-ben telik el, melyek bemenő karakterenként egyszer kerülnek hívásra, és még a főprogram is használ valamennyit. Egy

```
translit a—z A—Z
```

hívás esetén az idő 19%-át az **index**, 7%-át a főprogram és 4%-át az **xindex** emésztette fel. Amikor azonban a **to** halmaz kicsi, mint például

```
translit ¬@n @n
```

esetén, az időkiosztás a következő: **index** 5%, **xindex** és főprogram 6—6%. Tehát az **index** az a rutin, amelyet fel kell gyorsítani (ha B/K valaha is elég gyors tud lenni), mivel az általa felhasznált idő mind az input-karakterek számától, mind pedig a **from** halmaz méretétől függ.

Gyakorlatok

2.19 Írjuk le, milyen tevékenységet végeznek a következő parancsok:

```
translit a—b—d abcd
translit a—c xyz
translit —ac xyz
translit @@
```

Hogyan konvertálnánk szóközök sorozatát egyetlen szóközzé? □

2.20 A felhívójel egyik célja az, hogy olyan karakterek bevitelét oldja meg, amelyeket bonyolult gépelni, nehéz elolvasni, vagy speciális a jelentésük. Terjesszük ki az **esc**-t úgy, hogy elfogadja az **s**-et szóköznek, a **b**-t visszalépésnek, és esetleg oktális vagy hexadecimális számjegysorozatot tetszőleges bitsorozatnak! □

2.21 Az **error** elemi művelet felépítésekor használjuk fel **esc**-t oly módon, hogy egy felhívójel után álló pont egy tényleges pontnak az üzenetben való megjelenését eredményezze! □

2.8 SZÁMOK

A **translit**-et gyakran használjuk file-ok szűrésére, mielőtt azokban különböző dolgokat megszámlálnánk az előző fejezetben bemutatott **charcount** segítségével. Mivel akkor elsősorban a RATFOR bemutatására koncentráltunk, elhalasztottuk a **putdec** szubrutin ismertetését, amely a számlálás eredményét nyomtatta ki. Mielőtt tovább mennénk, bemutatjuk a **putdec**-et.

A **putdec(n, w)** az *n* számot írja ki egy legalább *w* karakterből álló sorozat formájában, amelybe negatív szám esetén az előjel is beleszámít. Ha *w*-nél kevesebb számú jelre van szükség, a sorozat balról szóközökkel töltődik fel. A *w*-nél hosszabb sorozatok automatikusan jönnek létre. Ez olyan tulajdonság, amely **putdec**-et hasznosabbá teszi, mint a legtöbb nyelvben előforduló szokványos kiviteli lehetőségek.

```
# putdec — decimális egész kiírása legalább w pozíción
   subroutine putdec(n, w)
   character chars(MAXCHARS)
   integer itoc
   integer i, n, nd, w

   nd = itoc(n, chars, MAXCHARS)
   for (i = nd + 1; i <= w; i = i + 1)
        call putc(BLANK)
   for (i = 1; i <= nd; i = i + 1)
        call putc(chars(i))
   return
   end
```

Az átalakítást a **putdec**-en belül az **itoc** végzi. Az **itoc** egy egész számot karakterekké konvertál a hívó által biztosított tömbben, és visszaadja a létrehozott karakterek számát az **EOS**-et nem beleszámítva. Az **itoc** a számjegyeket — mivel így egyszerűbb — fordított sorrendben állítja elő, majd mielőtt kilép, megfordítja a sorozatot.

```
# itoc — egész szám átalakítása karaktersorozattá
    integer function itoc(int, str, size)
    integer abs, mod
    integer d, i, int, intval, j, k, size
    character str(size)
    string digits "0123456789"

    intval = abs(int)
    str(1) = EOS
    i = 1
    repeat {                                  # számjegyek előállítása
        i = i + 1
        d = mod(intval, 10)
        str(i) = digits(d + 1)
        intval = intval / 10
        } until (intval == 0 | i >= size)
    if (int < 0 & i < size) {                 # előjel
        i = i + 1
        str(i) = MINUS
        }
    itoc = i — 1
    for (j = 1; j < i; j = j + 1) {           # sorrendfordítás
        k = str(i)
        str(i) = str(j)
        str(j) = k
        i = i — 1
        }
    return
    end
```

Az **abs** függvény saját paraméterének abszolút értékét képezi. (FORTRAN-ban az **iabs** használható.)

Az **itoc** inverze, melyet **ctoi**-nek nevezünk, egy karaktersorozatot alakít át egész számmá. Egy célszerű megoldás a következő:

```
n = ctoi(c, i)
```

hívás a **c** paraméter *i*-edik pozíciójánál kezdi a munkát. Kihagyja a szám előtti szóközöket és tabulátorokat, majd az ezek után következő számjegyeket a megfelelő numerikus értékre konvertálja. Az első, nem számjegy karakter előfordulásakor befejezi működését; kilépéskor *i* erre a pozícióra mutat. A kapott egész szám **n**-ben kerül tárolásra. A könyvben rendszeresen fogjuk **ctoi**-t használni.

```
# ctoi — karaktersorozat átalakítása egész számmá
    integer function ctoi(in, i)
    character in(ARB)
    integer index
    integer d, i
    string digits "0123456789"

    while (in(i) == BLANK | in(i) == TAB)
        i = i + 1
    for (ctoi = 0; in(i) ¬= EOS; i = i + 1) {
        d = index(digits, in(i))
        if (d == 0)                  #
            break
        ctoi = 10* ctoi+d — 1
        }
    return
    end
```

Most lássuk, mire jutottunk. A **charcount**-ot a **translit**-tel együtt használva sokféle hasznos információt tudunk nyújtani. Vezessünk be egy részleges job control nyelvet: két program soros összekapcsolására a | jelölést fogadjuk el. (Ne keverjük össze a logikai feltételekben szereplő „vagy" operátorral! Mivel túl sok jelre van szükségünk, ezért némelyiket időnként többféle értelemben is használnunk kell.) A

```
translit ¬@n | charcount
```

a következőt jelenti: „Vedd a **translit** által létrehozott eredményt mint a **charcount** bemenetét." Ezt a szerkezetet *csatlakozásnak* nevezzük. Az, hogy egy csatlakozás párhuzamosan működő folyamatok közötti közvetlen kapcsolat-e vagy annak az előírása, hogy milyen közbenső ideiglenes file-okat kell írni és olvasni, a legtöbb alkalmazás szempontjából lényegtelen. Számunkra az fontos, hogy mit tudunk ilyen csatlakozásokkal megoldani.

A fenti példában minden törlődik a szövegből, az újsor-karaktereket kivéve, melyeket azután megszámlálunk. A **linecount**-tal egyenértékű sorszámláló programot írtunk. A szavak számlálása egy kicsit bonyolultabb:

```
translit " @t@n" @n | translit ¬@n | charcount
```

Először minden szót külön sorban helyezünk el oly módon, hogy a nem látható karaktereket tömörítjük, azután megszámláljuk a sorokat, mint az előbb. Még a programban előforduló decimális számjegysorozatokat is meg tudjuk számlálni:

```
translit 0—9 9 | translit ¬9 | charcount
```

Ez minden számjegysorozatot egyetlen kilencessé konvertál, minden más karaktert töröl és megszámlálja a kilenceseket.

Dőreség lenne az utóbbi programhoz hasonlókat írni, mivel teljesen valószínűtlen, hogy efféle haszontalanságok gyűjteményét érdemes legyen létrehozni, hogy azután soha semmire ne használjuk őket. A **translit**-hez hasonló, hatékony eszközöknek viszont van értelmük. Használhatók egyedül vagy más szűrőkkel összekapcsolva (mint pl. a **charcount**), a legkülönbözőbb feladatok elvégzésére. Ez az eszközkészítés lényege.

Gyakorlatok

2.22 Sok számítógép az egészeket kettes komplemens formájában tárolja, ahol a negatív számok száma eggyel több, mint a pozitívaké. Más reprezentációkban a pozitív és negatív számok száma egyenlő ugyan, de a nullának kétféle ábrázolása van. Írjuk át a **putdec**-et úgy, hogy helyesen kezelje a kettes komplemens kódú aritmetikát! (Indokoljuk meg, miért nem jó erre a jelenlegi változat?) □

2.23 Módosítsuk **ctoi**-t úgy, hogy felismerjen egy nem kötelezően előírt + vagy − előjelet! □

2.24 Írjuk meg **ctof**-ot, amelyik egy karaktersorozatot lebegőpontos számmá konvertál. A rutinnak fel kell ismernie egy nem kötelezően előírt előjelet, egy nem kötelező tizedespontot, és a decimális exponens −1.23E−4 jelölésmódját. □

2.25 A **wordcount** közvetlenül számlálja meg a szavakat, ahelyett, hogy **translit**-et és **charcount**-ot használná. Megéri-e az elérhető nagyobb hatékonyság a többletmunkát? □

2.26 Írjunk egy csatlakozást, amely egy rejtjelezett file-t kisbetűssé alakít át (föltesszük, hogy a kulcs ismert). Képesnek kell-e lennie a **crypt**-nek karakterek fordítására, a **translit**-hoz hasonlóan? Kell-e a **translit**-nak ismernie a rejtjelezés, illetve a megfejtés módszerét? □

2.27 Írjuk meg a **tail** szűrőt, amely a bemenő szövegnek csak az utolsó *n* sorát adja eredményül, ahol *n* nem kötelező paraméter. Azaz

```
program | tail 10
```

a **program** által előállított szöveg utolsó tíz sorát nyomtatja ki. Természetesen a gyakorlati megvalósításkor *n*-hez egy felső határ tartozik. Ha nem adjuk meg *n* értékét, mi a kézenfekvő alapfeltételezés? □

2.28 Tervezzük meg és valósítsuk meg a **calc** szűrőt, amelyik egy zsebszámológépet szimulál! A **calc**-nak tartalmaznia kell legalább a +, − és = operátorokat, úgy, hogy olyan kifejezések, mint

```
1+2-3=
```

kiértékelhetők legyenek. A továbbfejlesztés tartalmazhat több operátort, zárójeleket, tárat stb. Hogy módosítanánk **calc**-ot, hogy egyetlen gépi szóban tárolható számoknál nagyobb értékeket kezeljen? □

2.29 Írjunk egy programot, amely tetszőleges bitsorozatot értelmez oktális vagy hexadecimális számként, karakterként és esetleg egy adott számítógép gépi utasításaként! A programot szűrőként valósítsuk meg. □

2.9 ÖSSZEFOGLALÁS

Az ebben a fejezetben bemutatott szűrők sokfélék, de a legtöbbjüknek van egy közös, alapvető tulajdonsága. A maga módján mindegyik azt sugallja, hogy a programok között átadott vagy a file-okban későbbi felhasználás céljára tárolt szövegek szabványos ábrázolásúak legyenek. Azáltal, hogy az egyes hardverkészülékek sajátságait olyan mértékben kiküszöböltük a programokból, amennyire csak lehetséges volt, megnöveltük a szabad együttműködésre alkalmas programok hatáskörét. Ugyanakkor a végső szűrők írásával, melyek ezen készülékeket szabványos formájú

file-okkal kapcsolják össze, a hardver-függő információt különválasztjuk és összegyűjtjük. Ennek következtében a gép perifériáiban alapvető változtatásokat hajthatunk végre anélkül, hogy ez egy-két programmodulnál többet érintene. Ez annak a lehetőségét is jelenti, hogy egy-egy új készülék használatba vétele kevés programozással megoldható. A programok továbbfejlesztése szempontjából ez igen lényeges kérdés.

A **translit** egy még általánosabb szerszám. Segítségével elfeledkezhetünk sok apró, a jelkészlettől függő problémáról, amelyek gyakran kísértenek a számítóközpontokban, és megnehezítik a szoftvercsomagok kombinált alkalmazását. A **translit**-ot arra használhatjuk, hogy vele két, nem tökéletesen együttműködő program között fennálló ellentéteket eltüntessünk, és ezáltal elkerülhetjük valamelyiknek vagy mindkettőnek az átírását. Emellett a **translit** önmagában is hasznos.

Ha egyszer már rájöttünk, hogy szűrők bevezetésével képesek vagyunk lokalizálásra és adaptálásra, akkor egyre inkább kézenfekvőnek találjuk majd meglevő programok kombinált felhasználását, az újak írása helyett. Nagyrészt elkerüljük azt a kísértést, hogy egy teljesen új csomagot hozzunk létre; ehelyett a már létező darabokat adaptáljuk. Egyszóval, egyre inkább eszközhasználóvá válunk.

BIBLIOGRÁFIAI MEGJEGYZÉSEK

A csatlakozás és a szűrők ötlete — egységes mechanizmus alkalmazása egyik program bemenetének egy másik program kimenetéhez való kapcsolására — M. D. McIlroy-tól származik. Kevésbé általános formában, „kommunikációs file-ok"-ként a Dartmouth Time-sharing System-ben is szerepel. A csatlakozások egy külösösen tiszta megvalósítását találjuk a UNIX operációs rendszerben, melyet K. L. Thompson és D. M. Ritchie fejlesztett ki a Bell Laboratóriumban. Itt a csatlakozó folyamatok párhuzamosan futnak, és megfelelő jelzők biztosítják, hogy a fogadó folyamat soha ne olvasson nem létező információt és a küldő folyamat soha ne állítson elő túl sok információt, mielőtt a fogadó valamennyit már el ne olvasott volna. A függőleges vonal az UNIX szintaxisa a csatlakozásra, Thompsontól származik. Lásd a „The Unix time-sharing system" c. cikket, CACM, 1974. július.

A **translit** alapelveit egy eredetileg M. D. McIlroy által írott programból vettük át.

Ha valaki a rejtjel-analízis bármely formája iránt érdeklődik, olvassa el David Kahn „The Codebreakers" c. könyvét (MacMillan, 1967).

3. fejezet

File-ok

Eddig olyan programokról beszéltünk, amelyek egyetlen bemenő jelsorozatot olvasnak és egyetlen kimenő jelsorozatot írnak anélkül, hogy sokat törődnének azzal, hogy ezek az adatok hogyan kerülnek a megfelelő helyre. Bár igen sok hasznos programnak valóban csak egyetlen inputra, ill. outputra van szüksége, ez általában mégsem elégséges. Ebben a fejezetben olyan programokat tárgyalunk, amelyek a környezetükkel bonyolultabb kölcsönhatásban vannak, több file-t írnak vagy olvasnak, valamint futásuk közben file-okat hoznak létre és szüntetnek meg. Ezek a programok főleg olyan operációs rendszer számára hasznosak, amely valamilyen állandó file-rendszert tartalmaz, ahol a futó programok számára könnyen hozzáférhető háttértárban hosszabb ideig lehet információt tárolni.

Sajnos, ilyen műveletekről még beszélni is nehéz, hogy a programozásukat ne is említsük, mivel:

— minden operációs rendszernek saját zsargonja van; nincs általánosan elfogadott terminológia;

— minden operációs rendszernek megvannak a saját lehetőségei és korlátai; nincsenek általánosan elfogadott funkciók;

— néhány dolog, amit meg akarunk valósítani, egyes operációs rendszerekben nem oldhatók meg egyszerűen; nincsenek tökéletes rendszerek.

Amint ezt az előző fejezetben is tettük, úgy próbáljuk elkerülni ezeket a nehézségeket, hogy tárgyalásunkat és programjainkat elemi műveletek köré építjük fel, olyan műveletek köré, mint a **gect**, a **putc** és a **getarg**, melyek logikája egyszerű, és mindegyikük egyetlen jól körülhatárolt feladatot hajt végre. Az elemi műveletek végrehajtásának módja és kifejezésük formája rendszerről rendszerre változni fog, de a funkciójuk könnyen érthető és megvalósítható lesz bármelyik rendszerben, ha ott eleve nem állnának már rendelkezésre. Elemi műveletek definiálása és alkalmazása abból a célból, hogy az operációs rendszertől való függőséget lokalizáljuk, a modularizáció egy alapvető formája. Ez az egyetlen módja annak, hogy valódi rendszerekkel megbirkózzunk.

3.1 FILE-OK ÖSSZEHASONLÍTÁSA

Az operációs rendszerrel való összetettebb kapcsolat egyik példája több inputfile vagy adathalmaz olvasása. A több bemenetű programok egyik példája az a program, amelyik két bemeneti forrást hasonlít össze, és kilistázza azokat a helyeket, ahol ezek különböznek. Ez hasznos eszköz, például az egyik program eredményeinek egy másik program outputjával való összehasonlítására, vagy egy szöveg két változatának

összevetésére. Ez a program olyan feladatot gépesít, amelyet túlságosan gyakran végeznek el manuálisan.

A **compare** tervezése attól függ, hogy mit kell összehasonlítani. Mi egy olyan változatot írunk, amely két szöveges file-t hasonlít össze. A további változatokat a gyakorlatokra hagyjuk. Szövegek esetében az összehasonlítás természetes egysége a sor — ha két sor hossza különböző, könnyen szinkronizálhatunk a következő sorban. Természetesen, ha az egyik file-ból egy sor hiányzik, ez lehetetlenné teszi a szinkronizációt az input hátralevő részében, ezért ez nem az elképzelhető legjobb összehasonlítás, de kezdetnek azért megfelel. A program vázlata a következő:

```
repeat {
        végy egy sort file 1-ből
        végy egy sort file 2-ből
        if (valamelyiken EOF volt)
                break
        if (line1 ¬= line2)
            a sorszám, a karakterpozíció és
            a hibás sorok nyomtatása
        }
if (csak az egyik input fejeződött be)
   üzenet, hogy melyik input fogyott el
```

Az általános vezérlési szerkezetet először úgy ellenőrizzük, hogy megvizsgálunk néhány kritikus határesetet. Ameddig a két file azonos egymással, nincs kimenő jelsorozat; ha egyszerre fejeződnek be, minden rendben van, még akkor is, ha mindketten üresek. Ha az egyik file a másik elejével egyezik, de annál rövidebb (a másik file prefixe), erről üzenet értesíti a felhasználót. Végül, ha a file-ok az egyes sorokon belül bárhol különböznek egymástól, ezek a sorok kerülnek kinyomtatásra.

Most kezdhetünk a részletekkel foglalkozni. Két sor összehasonlítása önálló feladat, amelyet egy külön rutinban kell elkülöníteni. Az **equal** függvény két jelsorozatot hasonlít össze: **YES**-t ad, ha azonosak, és **NO**-t ha különbözőek. Mindkét jelsorozatot természetesen **EOS** fejezi be. Azért hasonlítunk össze **EOS**-sel befejezett karaktersorozatokat és nem **NEWLINE**-nal lezárt sorokat, mivel az így kapott általánosabb rutin nem igényel többletmunkát, mégis nagyobb eséllyel lesz használható más programokban.

```
# equal — str1 és str2 összehasonlítása
    integer function equal(str1, str2)
    character str1(ARB), str2(ARB)
    integer i

    for (i = 1; str1(i) == str2(i); i = i + 1)
        if (str1(i) == EOS) {
            equal = YES
            return
            }
    equal = NO
    return
    end
```

A következő probléma, ami a jelen fejezet fő témája, a programnak a bemeneti forrásokhoz való kapcsolása — hogyan végezzük el a következő operációkat:

végy egy sort file 1-ből
végy egy sort file 2-ből

Mostanáig feltételeztük, hogy a programhoz alapfeltételezés szerint tartozik valamilyen standard input és standard output, amelyhez futás közben hozzárendelődik, és amelyeket a **getc** és a **putc** implicit módon használ. Ez gyakran egy kártyaolvasó és egy sornyomtató a kötegelt rendszerekben, vagy a felhasználói terminálbillentyűzet és nyomtató az interaktív környezetben. Majdnem minden operációs rendszer lehetőséget ad ezen alapfeltételezésként fennálló hozzárendelések megváltoztatására, és más be- és kiviteli berendezések használatára. Például a legtöbb kötegelt rendszerben használható olyan vezérlőkártya, amelynek hatása a következő:

Rendeld hozzá az **N** *FORTRAN logikai egység kódszámát az* **F** *külső file-hoz*

így a következő FORTRAN utasítások

read (N, ...)...

és

write (N, ...)...

a továbbiakban **F**-en fognak dolgozni. Interaktív rendszerekben gyakran szintén találunk ugyanilyen hatású parancsokat. PL/1-ben a következő utasításokban használt belső file-neveket:

get file (name)...

vagy

put file (name)...

hozzárendelhetjük a külső környezet valamilyen adatforrásához vagy kimeneti mechanizmusához. A vezérlőkártya szintaxisa rendszerenként nagyon különböző, de a lehetőség mindenütt rendelkezésre áll.

Így a **compare** egy kezdetleges változata két bemenő jelsorozatot tud olvasni, melyek belső neve megegyezés szerint például 1 és 2 FORTRAN-ban, **name1** és **name2** PL/1-ben, és amelyeket a felhasználónak kell a megfelelő külső file-okhoz rendelnie, vezérlőkártya vagy parancs segítségével. Ez a **compare** nem tartalmaz semmiféle explicit egymáshoz rendelést a file-ok és a külső adatforrások között — ezt a munkát valaki másnak kell elvégeznie.

```
# compare (egyszerű változat) — file1 és file2 összehasonlítása
   character line1(MAXLINE), line2(MAXLINE)
   integer equal, getlin
   integer lineno, m1, m2
```

```
    lineno = 0
    repeat {
        m1 = getlin(line1, INFILE1)
        m2 = getlin(line2, INFILE2)
        if (m1 == EOF | m2 == EOF)
            break
        lineno = lineno + 1
        if (equal(line1, line2 == NO)
            call difmsg(lineno, line1, line2)
        }
    if (m1 == EOF & m2 ¬= EOF)
        call remark("file1-nek vége")
    else if (m2 == EOF & m1 ¬= EOF)
        call remark("file2-nek vége)
    # különben megegyeznek
    stop
    end
```

A **remark** egy általános célú üzenetnyomtató. Ez tulajdonképpen a 2. fejezetben szereplő **error**, azzal az eltéréssel, hogy nyomtatás után visszatér a hívó programba, ahelyett hogy megállna. Figyeljük meg, hogy a **compare** nem ad üzenetet, ha a két file azonos. Egy program általában ne mondjon addig semmit, ameddig nincsen mondanivalója!

Ha valamilyen különbség mutatkozik a két file között, **difmsg** kinyomtatja a sorszámot és az eltérő sorokat; a részleteket ízlésünknek megfelelően magunk is kitölthetjük a **putc** és a **putdec** felhasználásával. Megoldásunk a következő:

```
# difmsg — eltérő sorok kiírása sorszámmal együtt
    subroutine difmsg(lineno, line1, line2)
    character line1(ARB), line2(ARB)
    integer lineno

    call putdec(lineno, 5)
    call putc(NEWLINE)
    call putlin(line1, STDOUT)
    call putlin(line2, STDOUT)
    return
    end
```

A

getlin (line, infile)

függvény hívás az **infile** belső nevű file következő sorát átmásolja a **line** karaktertömbbe. Biztosítja azt, hogy a bemenő sort egy újsor-karakter és egy **EOS** zárja le. A **getlin** **EOF**-et ad vissza, ha a file végét érzékeli, különben megadja a sor hosszát (az **EOS** nélkül). A **getc**-hez hasonlóan **getlin** a karaktereket belső ábrázolásra alakítja át. Az is elképzelhető, hogy a **getc**-nek egy változatát használja ennek elvégzésére, bár az gyakoribb, hogy a **getlin** az alacsonyabb szintű elemi művelet és **getc** hívja őt. (Az 1. fejezetben szereplő egyszerű **getc** és **putc** ténylegesen így dolgozik.)

Mivel **getlin** vagy egy karaktersorozat hosszát vagy **EOF**-et ad vissza, még egy érvünk van amellett, hogy **EOF**-et célszerű negatív értéknek választani.

A **putlin** olyan elemi művelet, amely egy sort kivisz egy adott file-ba. Ez tulajdonképpen **getlin** komplemense és végrehajtja a szükséges jelátalakításokat. Feltesszük, hogy **putc** és **putlin** egymást időben átfedő hívásai során keletkező kimenet, a megfelelő sorrendben jelenik meg. Az **STDOUT** a standard output belső neve; amint ez várható, van egy **STDIN** is, a standard input számára. A rendszernek kell arról gondoskodnia, hogy ezek a file-ok olvasásra és írásra készen álljanak, amikor a program futása elkezdődik.

Gyakorlatok

3.1 Valósítsuk meg **getlin**-t és **putlin**-t. Változtassuk meg az 1. fejezetben szereplő **getc**-t és **putc**-t úgy, hogy felhasználják **getlin**-t és **putlin**-t. Gondoskodjunk arról, hogy **getc** és **getlin** az ugyanazon, illetve a különböző file-okra vonatkozó, időben egymást átfedő hívásai jól működjenek. Végezzük el ugyanezt **putc**-re és **putlin**-re. □

3.2 Nem interaktív környezetben a **compare**-nek valószínűleg nem célszerű túl sokat kinyomtatnia egymástól nagyon különböző file-okról. Vezessünk be egy nem kötelező paramétert abból a célból, hogy az összehasonlítás érjen véget adott számú eltérő sor előfordulása után. Egy másik lehetőség, hogy **compare** megáll az első eltérés után. Ebben az esetben a paraméter egynél több eltérést engedhet meg. □

3.3 Hozzuk létre a **compare**-nek egy olyan változatát, amelyben két sort azonosnak tekintünk, ha mindegyikükben a szóközökből álló sorozatokat egyetlen szóközzel helyettesítve azonosakká válnak. El tudjuk-e érni ezt az eredményt a **translit** használatával? Át tudunk-e venni programrészeket a **translit**-ból?

3.4 (Nehéz) Jelenlegi **compare**-ünk nagyon kezdetleges. Annak a helynek a megtalálására használható, ahol két file különbözővé válik, de ezután hamarosan csődöt mond, és igen terjedelmes, de kis információtartalmú outputot ad. Találjunk ki olyan módszert, amely alkalmas hiányzó és máshová helyezett sorok kezelésére! Mennyi helyet és időt igényel a módszer a file-ok méretének függvényében? Van-e olyan eset, amikor teljesen csődöt mond? (Lásd a fejezet végén levő bibliográfiai megjegyzéseket.)

3.2 FILE-OK NÉV SZERINTI EGYMÁSHOZ RENDELÉSE

Térjünk vissza ahhoz a kérdéshez, hogy egy file külső nevét hogyan tudjuk a programban használt névhez rendelni. A külső név általában a file-nak egy filerendszerben használt neve vagy valamilyen ideiglenes file neve, amelyet csak a job idejére jelöltek ki. Ritkábban előfordul az is, hogy a külső név egy B/K készülék, például egy mágnesszalag. Mint már említettük, a külső és a belső név egymáshoz rendelésének hagyományos módja egy vezérlőkártya használata. Bizonyos környezetekben ez az egyetlen lehetőség. Ugyanakkor senki sem szeretné azt írni, hogy

```
connect file1 to name1
connect file2 to name2
compare
```

csak azért, mert a **compare** csak az 1-es és a 2-es belső nevekkel tud dolgozni. Még csak nem is a **connect** parancsok előkészítéséhez szükséges többletmunka az, ami a fenti megoldást ellenszenvessé teszi — bár ez is elég kellemetlen. A baj ott van, hogy emlékeznünk kell a program által használt belső nevekre. Mennyivel természetesebb lenne azt írni, hogy

compare file1 file2

és a **compare**-re bízni a file-okhoz való hozzáférést!

Tegyük föl, hogy a tényleges file-nevet átadjuk a programnak és az sajátmaga intézi el a külső és a belső nevek közötti megfeleltetést. Ez más szóval azt jelenti, hogy a program futás közben végzi el azt, amit a vezérlőkártyák hajtottak volna végre a végrehajtás előtt. Nevezzük azt az elemi műveletet, amely ezt a feladatot végrehajtja, **open**-nek. Az **open** mindent elintéz, ami a file eléréséhez szükséges, és kijelöl egy belső file-nevet vagy számot, amelyet függvényértékként visszaad:

belső név= **open** (*külső név, elérési mód*)

A következő hívásokban **getlin** és **putlin** ezt a *belső nevet* használja.

Az adott környezettől függően az **open** a külső néven kívül más információt is igényelhet — puffer-terület, elérési mód stb. A külső néven kívül az összes információt a továbbiakban az *elérési mód*ban egyesítjük, amely programjainkban mindig a **READ, WRITE** vagy **READWRITE** szimbolikus konstansok egyike lesz, és azt mutatja, hogyan akarja a program a file-t használni. Természetesen, ha az általunk használt rendszer valamilyen file-védelmet nyújt, az **open** gondjaira van bízva az általunk kívánt elérés megszervezése.

Az **open** mindenféle hibát úgy jelez, hogy az **ERR** értéket adja vissza egy szabályos belső név helyett. Egyik kényelmes megvalósítás kis pozitív számokat használ belső neveknek; az **open** ilyenkor egyszerűen a legelső, még fel nem használt értéket adja vissza. Ebben az esetben **ERR** értéke célszerűen nulla vagy negatív.

Az **open** birtokában **compare** a következőképpen írható meg:

```
# compare — két file összehasonlítása
   character arg1(MAXLINE), arg2(MAXLINE)
   character line1(MAXLINE), line2(MAXLINE)
   integer equal, getarg, getlin, open
   integer infil1, infil2, lineno, m1, m2

   if (getarg(1, arg1, MAXLINE) == EOF
     | getarg(2, arg2, MAXLINE) == EOF)
       call error("hívás: compare file1 file2.")
   infil1 = open(arg1, READ)
   if (infil1 == ERR)
       call cant(arg1)
   infil2 = open(arg2, READ)
   if (infil2 == ERR)
       call cant(arg2)
   lineno = 0
   repeat {
       m1 = getlin(line1, infil1)
```

```
        m2 = getlin(line2, infil2)
        if (m1 == EOF | m2 == EOF)
            break
        lineno = lineno + 1
        if (equal(line1, line2) == NO)
            call difmsg(lineno, line1, line2)
        }
    if (m1 == EOF & m2 ¬= EOF)
        call remark("file1-nek vége")
    else if (m2 == EOF & m1 ¬= EOF)
        call remark("file2-nek vége")
    stop
    end
```

A **cant(name)** kinyomtatja a következőt:

```
name: nem lehet megnyitni
```

és kilép. Bár ez csupán egy jelentéktelen feladat, elég gyakran fordul elő ahhoz, hogy megérdemeljen egy külön rutint.

A legtöbb interaktív rendszer lehetőséget ad a hozzárendelésre (az elegancia különböző fokain); kötegelt környezetben viszont ez kevésbé elterjedt. Nyilvánvaló, hogy a fenti funkciónak léteznie kell minden olyan operációs rendszerben, amelyben file-ok név szerinti tárolására lehetőség van, különben hogyan működhetne a parancsnyelvet értelmező program? Ugyanakkor nagyon is gyakran a műveletet önkényesen „rendszerprogramokra" korlátozzák és megtiltják a közönséges felhasználóknak. Ez sajnálatos, mivel igen fontos, hogy egy program képes legyen a bemeneti forrásaihoz dinamikusan kapcsolódni. A jó, emberi szemléletű tervezés eredményeként a programok használata könnyű lesz. Mi mondjunk csak annyit, hogy „hasonlítsd össze ezeket meg ezeket a file-okat" és a program végezzen el minden munkát. Minél kevesebb előkészületre van szükség egy program tényleges használata előtt, annál valószínűbb, hogy eszközként fogjuk azt számontartani.

Mit tudunk tenni, ha a dinamikus file-hozzárendelés nem megvalósítható? Egyik lehetőség annak rögzítése, hogy az inputfile-ok konvenció szerint standard helyeken vannak, mint például az 1, 2, 3, ... bemeneti sorozatszámok. Az inputfile-ok számát megadhatja a felhasználó (ez a kevésbé vonzó megoldás) vagy kikövetkeztetheti a program.

A másik lehetőség, hogy kiválasztjuk a saját belső neveinket, amelyet a programnak paraméterként átadunk. Ezután azt írjuk, hogy

```
connect file1 2
connect file2 3
compare 2 3
```

Ekkor az egymáshoz rendelést mi végezzük el ugyan, de azt legalább nem kell megjegyeznünk, hogy a program milyen neveket használ.

Egy végső lehetőség, amely az adott operációs rendszer részéről némi együttműködést igényel, az, hogy a file-okat név szerint specifikáljuk, majd egy programot

használunk a megfelelő vezérlőkártyák előállítására, amely a külső és belső nevek közötti megfeleltetést tartalmazza, és utolsó lépésként meghívja **compare**-t. Ezután le kell futtatni a vezérlőkártyákat, célszerűen ugyanazon job részeként, amely előállította őket.

Gyakorlatok

3.5 Működik-e a

compare f f

az Ön gépén? Ha nem, miért nem? Kellene-e működnie? Miért lehet szükség ilyen műveletre?

3.6 Módosítsuk **compare**-t úgy, hogy ha egyetlen paraméterrel hívjuk meg, feltételezi, hogy a másik file a standard input. Ezután használhatjuk a **compare**-t olyan csatlakozásokban, mint

expand | compare f

Ér-e ez a lehetőség valamit is, ha a csatlakozásokat úgy kell megvalósítanunk, hogy ideiglenes file-okat használunk?

3.7 (Nehezebb) Terjesszük ki a csatlakozások szintaxisát és szemantikáját oly módon, hogy a **compare** mindkét forrás-file-ja programokból származó output lehessen!

3.3 FILE-BEILLESZTÉS

Az **open** birtokában kényelmesen tudunk olyan eszközöket létrehozni, melyek tetszőleges számú file-t használnak. Az egyik ilyen az **include**, amely standard inputját átmásolja standard outputjára, kivéve azokat az input sorokat, melyek az

include filename

szavakkal kezdődnek, mert ezeket a **filename** nevű file teljes tartalmával helyettesíti.

A könyvben szereplő sok nagyobb program használja az **include**-ot, leggyakrabban **common** deklarációsorozatok beillesztésére. Ez biztosítja, hogy egy **common** blokkot használó összes rutinba ugyanaz a változat kerül, és csak egyetlen változtatás szükséges az összes rutin módosításához. Hasonlóképpen, egy program fordításakor a szimbolikus konstansok definícióinak beillesztése is egy **include** sorral történik. Mivel amikor csak lehet, szimbolikus konstansokat használunk, programjaink legtöbbjébe beillesztjük az **EOF**-hez és **EOS**-hez hasonló fogalmak egy standard halmazát, és esetleg egy, az adott programokra specifikus másik halmazt is. Az **include**-ot használhatjuk még előre elkészített vezérlőkártyák vagy — program kipróbálása esetén — tesztsorozatok beillesztésére; ilyenkor a beillesztett file-ok előre kidolgozott teszteseteket tartalmaznak. (Az eredményeket a **compare**-rel lehet majd összehasonlítani!)

Az **include** a PL/1 előfeldolgozóban található lehetőségnek egy általánosítása. A

% include file;

alakú sorokat a file tartalma helyettesíti. A mechanizmus azonban csupán PL/1 forrásprogramokra korlátozódik, ezért másra nemigen alkalmas. A mi változatunk ezzel szemben bármilyen szöveg-file-re használható.

Az **include** vázlatos felépítése a következő:

```
while (getlin(line, file) ¬= EOF)
    if (a sor elején "include" áll)
        az új file beillesztése
    else
        a sor kivitele
```

Ha a beillesztendő file további **include**-ot tartalmazhat, akkor nyilvánvalóan rekurzív eljárásról van szó, azaz a műveletet önmaga felhasználásával definiáljuk. Ha az általunk használt nyelv nem engedi meg a rekurziót (mint pl. a FORTRAN), akkor vagy egy, a file-azonosító számára használatos verem segítségével szimulálhatjuk azt, vagy végképpen le kell mondanunk róla. Az egymásba skatulyázott **include**-ok hasznosnak bizonyulnak, és kezelésük egyáltalában nem nehéz.

A másik kérdés annak eldöntése, hogy egy sor tartalmaz-e *include*-ot. Ezt célszerű két lépésben végrehajtani: megkeresni a sor első szavát és összehasonlítani az *include* szóval. Ebből a célból megírjuk a **getwrd** rutint, amelyik leválasztja a bemenő sor következő szavát, ahol a „szó" a szóköztől különböző olyan jelek sorozata, amelyeket szóközök, tabulátorok vagy újsor-karakterek zárnak le. A **getwrd** átlépi az esetleges szóeleji szóközöket és tabulátorokat, visszaadja a szót és a szó hosszát (amelyet ezúttal nem használunk). A **getwrd** egy indexet is beállít, amelyik a szó utáni első karakterre mutat, azért, hogyha a szó *include*, újra használhassuk a **getwrd**-t a file nevének kiolvasására.

```
# getwrd — egy nem üres szó átvétele in(i)-ből out-ba i növelésével
    integer function getwrd(in, i, out)
    character in(ARB), out(ARB)
    integer i, j

    while (in(i) == BLANK | in(i) == TAB)
        i = i + 1
    j = 1
    while (in(i) ¬= EOS & in(i) ¬= BLANK
       & in(i) ¬= TAB & in(i) ¬= NEWLINE) {
        out(j) = in(i)
        i = i + 1
        j = j + 1
        }
    out(j) = EOS
    getwrd = j — 1
    return
    end
```

A fő rutin a következő:

```
# include — "include" file helyettesítése a file tartalmával
    character line(MAXLINE), str(MAXLINE)
    integer equal, getlin, getwrd, open
    integer infile(NFILES), len, level, loc
    string incl "include"

    infile(1) = STDIN
    for (level = 1; level > 0; level — 1) {
        while (getlin(line, infile(level)) ¬= EOF) {
            loc = 1
            len = getwrd(line, loc, str)
            if (equal(str, incl) == NO)
                call putlin(line, STDOUT)
            else {
                level = level + 1
                if (level > NFILES)
                    call error("Az include-ok skatulyázása túl mély.")
                len = getwrd(line, loc, str)
                infile(level) = open(str, READ)
                if (infile(level) == ERR)
                    call cant(str)
                }
            }
        if (level > 1)
            call close(infile(level))
        }
    stop
    end
```

Az **equal** dönti el, hogy a szó *include*-e; ezt a rutint a fejezet előző részében már megírtuk. Vegyük észre, hogy az **equal**-t két sorozatnak és nem két sornak az összehasonlítására használtuk. Az általánosság máris kifizetődött!

A vázlatos megoldásunkhoz képest az **if—else** szerkezetet megfordítottuk, mivel úgy találtuk, hogy jobb a rövidebb programdarabot kapcsolni az **if**-hez és megtartani a hosszabbat az **else** részére. Ily módon nem veszítjük szem elől a lap aljára kerülő rövid **else**-bejegyzést.

A **close** alapművelet az **open** fordítottja: megszünteti a külső és a belső név egymáshoz rendelését, és felszabadítja a belső nevet és az ehhez kapcsolódó erőforrásokat más felhasználások számára. (A **close** *nem* azonos a FORTRAN-programozók számára ismerős **rewind** művelettel, amely a file elejére áll lezárás nélkül.) Azért használjuk itt a **close**-t, mert nem tudjuk sem azt, hogy egy adott file hányszor kerül felhasználásra, sem azt, hogy hány különböző file-ra lesz szükség. Mivel a legtöbb rendszerben az egyidejűleg nyitott file-ok száma korlátozott, explicit módon le kell zárnunk a file-okat, hogy ki ne fogyjunk a belső nevekből.

Bár programjaink gondosan ügyelnek arra, hogy lezárják a befejezett file-okat, célszerű, hogy a rendszer a program befejeződésekor minden nyitott file-t lezárjon. Ez egyszerűsíti az abnormális programbefejeződések kezelését.

Az **include** feltételezi az **open**-nek egy eddig nem említett implicit tulajdonságát — egy file a megnyitásakor a kezdetéhez van pozicionálva. Ez a tulajdonság az **include** számára alapvető fontosságú, hiszen egy file általában többször kerül beillesztésre.

Gyakorlat

3.8 Módosítsuk az **include**-ot egy olyan rendszer számára, ahol nem lehet file-okat név szerint megnyitni! Írjuk le a felhasználás módszerét! □

3.4 FILE-EGYESÍTÉS (KONKATENÁCIÓ)

Következő programként a **concat**-ot írjuk meg, amelyik név szerint megadott inputfile-ok halmazát egyesíti a standard outputon. A **concat** egy gyakori alkalmazása különálló file-ok egyesítése abból a célból, hogy egy olyan másik program használja őket, amelyik csak a standard inputról tud olvasni, mint pl. a 2. fejezetben szereplő szűrők. Ugyanakkor ez a legegyszerűbb módja a file kinyomtatásának formai átalakítás vagy egyéb értelmezés nélkül.

Valójában a **concat** az *include* egy változata, amely az összes file nevét egy paraméterlistából veszi és nem a beillesztést kiváltó **include** parancsokból. Ez könnyebbé teszi sokoldalú felhasználását, mivel nem kíván előkészítést. A programot sem nehéz megírni:

```
# concat — file-ok egyesítése a standard outputon
   character name(NAMESIZE)
   integer getarg, open
   integer fin, i

   for (i = 1; getarg(i, name, NAMESIZE) ¬= EOF; i = i + 1) {
        fin = open(name, READ)
        if (fin == ERR)
             call cant(name)
        call fcopy(fin, STDOUT)
        call close(fin)
        }
   stop
   end
```

Figyeljük meg, ha nincsenek inputfile-ok, a **concat** a megfelelő módon viselkedik — nem állít elő eredményt.

A tényleges másolást az **fcopy** végzi. Az **fcopy** természetesen lényegében azonos azzal a programmal, amellyel az 1. fejezetben a tárgyalást elkezdtük, csak szubrutin formába öntöttük. Az egyetlen különbség az, hogy a program argumentumként megadott file-okat olvas és ír a standard input és standard output használata helyett.

```
# fcopy — filemásolás in-ből out-ba
   subroutine fcopy(in, out)
   character buf(MAXLINE)
   integer getlin
   integer in, out

   while (getlin(buf, in) ¬= EOF)
        call putlin(buf, out)
   return
   end
```

Az **fcopy** feltételezi, hogy minden file nyitott, pozicionált és feldolgozásra készen áll; ő egyszerűen csak másol. Ily módon felhasználhatjuk file-ok részeinek a másolására is. Ha az **fcopy** gondosan megnyitná és lezárná a file-okat, felhasználhatósága korlátozott lenne. Ne építsünk programjainkba önkényes megszorításokat, különösen ne úgy, hogy túl sokféle tevékenységet próbálunk elvégeztetni velük. Ha file-okat akarunk megnyitni és lezárni, építsünk egy újabb réteget az **fcopy** fölé, ahogy mi az előzőkben eljártunk.

3.5 FILE-LISTÁZÓ

A határozatlan számú bemenettel és egyetlen kimenettel rendelkező programok közül az egyik leghasznosabb a file-nyomtató vagy -listázó program. A **print**-et egy vagy több file-név paraméterrel hívjuk meg, a program kinyomtatja a file-okat fölső és alsó lapszélbeállítással és minden egyes lap tetejére kiírja a file nevét, a lapszámot és esetleg a dátumot és az időt. Minden új file új lapon kezdődik. A **print**-et a **concat** helyett használhatjuk, amikor tetszetős külsejű, önmagát azonosító listát akarunk file-okról készíteni.

A program vázlatos felépítése a következő:

```
minden file-ra
    olvasd be a file nevét; legyen ez name
        open(name)
        fprint(name, fin)
        close(fin)

fprint(name, fin)
    kezdőérték-beállítás
        while (getlin(line, fin) ¬= EOF) {
            if (a lap teteje)
                fejlécnyomtatás
            a sor nyomtatása
            if (a lap tele van)
                soremelések a lap aljáig
            }
        if (a lap nincs tele)
            soremelések a lap aljáig
```

Ez a szervezés a paraméterlista feldolgozását az egyik szintre koncentrálja, az egy-egy file-ra vonatkozó, a sorszámlálással kapcsolatos összes részletet pedig egy alacsonyabb szinten gyűjti össze. A **print** tényleges programja a következő:

```
# print — file-ok listázása fejléccel
    character name(NAMESIZE)
    integer getarg, open
    integer fin, i
```

```
for (i = 1; getarg(i, name, NAMESIZE) ¬= EOF; i = i + 1) {
     fin = open(name, READ)
     if (fin == ERR)
          call cant(name)
     call fprint(name, fin)
     call close(fin)
     }
stop
end
```

Az **fprint**-et alaposan át kell gondolnunk, hogy a határesetekben ne mondjon csődöt. A legnyilvánvalóbb hibalehetőség az, ha nem megfelelő számú sort nyomtatunk minden oldalra, mert ilyenkor az output fokozatosan felhalmozódik vagy lecsökken az egymásután következő lapokon. Kevésbé nyilvánvaló az a probléma, ha egy file pontosan kitölti az utolsó oldalt, mert ilyenkor a következő file-t a következő lap tetején kell kezdeni, egy üres lap (vagy ami még rosszabb: egy fejléccel ellátott üres lap) létrehozása nélkül.

```
# fprint — a "name" nevű file listázása fin-ből
   subroutine fprint(name, fin)
   character line(MAXLINE), name(NAMESIZE)
   integer getlin
   integer fin, lineno, pageno

   pageno = 0
   lineno = 0
   while (getlin(line, fin) ¬= EOF) {
        if (lineno == 0) {
             call skip(MARGIN1)
             pageno = pageno + 1
             call head(name, pageno)
             call skip(MARGIN2)
             lineno = MARGIN1 + MARGIN2 + 1
             }
        call putlin(line, STDOUT)
        lineno = lineno + 1
        if (lineno >= BOTTOM) {
             call skip(PAGELEN-lineno)
             lineno = 0
             }
        }
   if (lineno > 0)
        call skip(PAGELEN-lineno)
   return
   end
```

A **MARGIN1** és **MARGIN2** szimbolikus konstansok a fejléc előtti és utáni sorok számát jelölik. A lapon levő utolsó szövegsor sorszáma a **BOTTOM** szimbolikus konstans; **PAGELEN** az egy lapra kerülő sorok száma. A szabványos 8,5×11 hüvelyk (kb. 21,6 cm×28 cm) méretű papíron hüvelykenként 6 sort számolva,

PAGELEN értéke 66. A lapszélek rendszerint egyenként két vagy három sor szélesek.

A **skip** *n* számú sort emel; erre a célra inkább egy apró külön rutint használunk, mintsem hogy zavarossá tegyük **fprint**-et a ciklus négyszeri leírásával.

```
# skip — n üres sor kivitele
  subroutine skip(n)
  integer i, n

  for (i = 1; i <= n; i = i + 1)
       call putc(NEWLINE)
  return
  end
```

Mivel azt akarjuk, hogy a listán a file-ok neve jelenjen meg, természetes és kényelmes, ha azokat valamilyen vezérlőkártyás megoldás helyett dinamikusan, név szerint rendeljük a programhoz. A **head** szubrutin egyetlen sort nyomtat ki a file nevével és a lapszámmal. A dátumot és az időt szintén kiírhatja, ha ezek ismertek.

```
# head — fejrésznyomtatás
  subroutine head(name, pageno)
  character name(NAMESIZE)
  integer pageno
  string page "Page"

  call putlin(name, STDOUT)
  call putlin(page, STDOUT)
  call putdec(pageno, 1)
  call putc(NEWLINE)
  return
  end
```

Mivel a **putlin** kivitelkor nem tesz a karaktersorozat végére újsor-karaktert, egy kimenetsor felépítéséhez **putlin**-t többször is hívhatjuk, amint ezt a névnek és a lapszámnak ugyanazon sorban való kiírásakor tettük.

Ha az alapeszköz már működik, sokféleképpen finomíthatjuk, és ezek közül néhányra nagy szükség is van. A következő lehetőségeket vehetjük figyelembe:
— tabulátoroknak szóközökre való konverziója
— a lapméretre, a bekezdési mélységre, a sortávolságra, a tabulátorpozíciókra stb.-re vonatkozó alapfeltételezések megváltoztatása
— hosszú sorok kezelése
— sorok számozása
— adott lapokon való indulás és megállás
— több file párhuzamos listázása
— többhasábos nyomtatás.

Ezek mindegyike egyszerű, az utolsó többhasábos nyomtatástól eltekintve. Ezt természetesen meg lehet oldani oly módon, hogy a teljes oldalt összegyűjtjük a nyomtatás előtt.

Egy program írásakor nagy a kísértés, hogy újabb és újabb, a fentiekhez hasonló „szolgáltatás"-okat építsünk bele, apró dolgokat, amelyeket a program megtesz a

számunkra. De legyünk óvatosak! Ha ezek a szolgáltatások nem működnek együtt egy egységes terv szerint, az eredmény nem lesz egyéb egymástól független funkciók kusza halmazánál, melyek legtöbbjét azért nem fogják használni, mert senki sem tudja majd fejben tartani őket. Ha már a legegyszerűbb alkalmazásoknál is utána kell néznünk egy program használati módjának, már tudhatjuk, hogy túl messzire mentünk. Ha kétségeink vannak, tekintsük a „szolgáltatásokat" hátrányos tulajdonságoknak. (Gondoljunk a százpengéjű svájci katonai késre, melynek használatát éppen sokoldalúsága nehezíti meg!)

Nem kötelező paraméterek megadására a következő szintaxist találjuk kényelmesnek: legyenek ezek a paraméterek egybetűsek vagy álljanak csupán néhány karakterből. Olyan karakter vezesse be őket, amelyik nem állhat egy file-név elején, így a paramétereket meg lehet különböztetni a file-nevektől. (Mi kötőjelet használunk.) Ha például a **print** lehetőséget ad többhasábos kimenet készítésére, ezt a **—c***n* paraméterrel jelöljük, ahol *n* a hasábok száma.
Így

```
print —c4 file1 file2...
```

négyhasábos formában nyomtatja ki a file-okat. Ha a paramétereket szigorúan balról jobbra dolgozzuk fel, megvan annak a lehetősége, hogy az egyik file-t adott paraméterek szerint nyomtassuk ki, majd a paraméterek közül egyet vagy többet megváltoztassunk a következő file számára.

Ha szabadon választható szolgáltatásokat nyújtunk, felmerül a kérdés, hogy mi történjen akkor, ha valamelyik lehetőség meghatározatlan marad. Ezt *sohasem* szabad hibának tekinteni; ehelyett valamilyen alapfeltételezéssel kell élni. A program alapfeltételezéseinek helyes megválasztása egyszerű dolognak tűnhet, de ha rosszul oldjuk meg, az mindenkit zavar (vagy a programot nem fogják használni). A döntés néha nyilvánvaló: a lapméretek például lényegében szabványosak. Más esetekben azonban kevésbé tiszta a helyzet: alapfeltételezésként széttördelje-e a **print** a túl hosszú sorokat több, rövidebb sorra? Ha igen, hol? Ne feledjük, hogy eszközöket készítünk és tegyük ezeket olyan hasznossá és kényelmessé, amennyire ez csak lehetséges. Egyik alapelv az, hogy amit gyakran kell mondani, az tömör legyen; ezért az alapfeltételezéseknek a leggyakoribb felhasználást kell tükrözniük. Ezen túlmenően, az alapfeltételezéseket úgy kell megválasztani, hogy az a felhasználó is elfogadható eredményeket kapjon, aki nem ismeri a lehetőségeket. Próbáljuk minél kevésbé meglepni a felhasználókat, és ne korlátozzuk lehetőségeiket!

Gyakorlat

3.9 Valósítsuk meg a **print** néhány továbbfejlesztését, melyekben nem kötelező paramétereket használhatunk! Mielőtt hozzáfognánk, próbáljuk előre jelezni, hogy melyiket fogják gyakran használni és melyiket egyáltalán nem. Miután az új változatot használtuk egy ideig, vizsgáljuk meg, mennyire váltak be jóslataink. A **print** mely lehetőségeit kell alapfeltételezésként megvalósítani? □

3.6 TÖBBLÉPCSŐS FELDOLGOZÁS: CSATLAKOZÁSOK

Ez a könyv eszközökről szól, s az olvasó most már valószínűleg rájött arra, hogy egy kis munkával a **print**-et eszközként használhatja bármilyen program eredményeinek kinyomtatására. Ha ez könnyen menne, akkor más programoknak soha nem kellene olyan részeket tartalmaznia, mint a többhasábos nyomtatás — a **print** valamelyik változata minden felhasználót ki tudna elégíteni.

Tegyük föl, hogy módosítjuk egy kissé a **print**-et oly módon, hogy ha nem kap file-neveket paraméterként, akkor bemenetét a standard inputról vegye. A program eredeti szerkezetének köszönhetően ez a feladat egyszerű. Csupán egy vizsgálatot kell beépítenünk a paraméterek hiányára vonatkozóan és egy üres file-nevet, amely a standard inputot jelenti.

```
# print (alapfeltételek szerint bemenet STDIN) — file-ok nyomtatása
#  fejléccel
   character name(NAMESIZE)
   integer getarg, open
   integer fin, i
   string null ""

   for (i = 1; getarg(i, name, NAMESIZE) ¬= EOF; i = i + 1) {
        fin = open(name, READ)
        if (fin == ERR)
             call cant(name)
        call fprint(name, fin)
        call close(fin)
        }
   if (i == 1)      # egyik file sem volt megnevezve
        call fprint(null, STDIN)
   stop
   end
```

Bármely program, amelyik a **print**-et akarja utófeldolgozóként használni, elérheti ezt csupán azzal, hogy kimenetét a **print** standard inputjára irányítja. A **print**-nek magának nem kell tudnia, vagy törődnie azzal, hogy ezt más program használja. A 2. fejezetben bevezetett csatlakozási jelöléssel pl. egyszerűen azt mondhatjuk, hogy

```
program... | print
```

Ez bármely program számára lehetővé teszi, hogy formázott kimenetet adjon a **print** lehetőségein belül. És mivel a **print**-et széleskörűen fogják használni, érdemes némi munkával szolgáltatásait és hatékonyságát javítani.

A csatlakozások megoldása rendszerről rendszerre némileg változik. Ideális esetben **program** és **print** a rendszer által összekapcsolt konkurrens folyamatok, és egyikük sem tudja, hogy a másik fut. Kevésbé kívánatos, de minden bizonnyal megoldhatóbb az ideiglenes file-okon keresztül való kommunikáció. Vezessük be a

```
prog >file
```

jelölést arra, hogy **prog** standard outputja a **file**-ban gyűlik össze, melynek létrehozása is megtörténik, ha ez szükséges (célszerű, ha ezt a rendszer automatikusan és a program tudta nélkül elvégzi). Ekkor a csatlakozást a következőképpen szimulálhatjuk:

```
prog >tempfile
print <tempfile
remove tempfile
```

ahol **<file** analóg módon a standard inputnak a **file**-ból mint forrásból való vételét jelenti, a **remove** pedig a **tempfile** megszűnését idézi elő. Az ideiglenes file-ok nehézkesebbek, mint a közvetlen csatlakozások, de el lehet boldogulni velük, különösen ha operációs rendszerünk jól támogatja file-ok kívánság szerinti létrehozását.

A hatékonyság problémáira érzékeny olvasók eltűnődhetnek, hogy vajon gazdaságos-e két program használata ott, ahol egy is elég lenne. A válasz a tényleges költségektől függ, amelyeket gyakran helytelenül becsülnek föl. A legtöbb ember, aki „hatékonyság"-ról beszél, elsősorban abban érdekelt, hogy mennyi gépi erőforrást használnak fel a végső futás során vagy az emberi beavatkozást nem igénylő termelésben, és nem törődik avval, hogy mennyi „emberi" és gépi időt használtak fel a fordításoknál, hibakeresésnél és egyéb zsákutcákban, melyek a végső futást készítették elő. Könyvünkben azon az állásponton vagyunk, hogy az emberek sokkal többe kerülnek, mint a gépek, és ez az aránytalanság a jövőben még növekedni fog. Ezért legfontosabbnak azt tekintjük, hogy az emberek a munkájukat a lehető legkevesebb nehézséggel és legkisebb erőfeszítéssel végezhessék el.

Ezt az elvet az eszközök biztosításával is támogathatjuk. Nem elegendő azonban a sok „segédprogram", ha ezek használata nehézkes, csak egy-egy speciális esetet kezelnek (még ha sok „szolgáltatást" nyújtanak is), és nem kapcsolhatók össze hasznos módon más eszközökkel. Az eszközöknek együtt kell működniük. A csatlakozások egyik előnye, hogy a programkészítéssel foglalkozókat arra ösztönzi, hogy szem előtt tartsák programjaiknak más programokhoz kapcsolását. Ez bizonyos mértékig elősegíti a szabványosítást, mivel nem hozhatók létre csatlakozások olyan programmal, amelynek más programokhoz való kapcsolódása nem egyértelmű.

A csatlakozások érdekében felhozható másik érv az, hogy egyszerűbb tevékenységek elvégzésére alkalmas kisebb programok létrehozására ösztönöz. A kisebb programokat sokkal könnyebb megírni, belőni, dokumentálni, karbantartani és önállóan továbbfejleszteni, mint ha egyetlen monstrumot állítanánk össze belőlük. És természetesen különálló programoknak sokféle újszerű kombinációja képzelhető el, ami aligha lehetséges, ha ezeket már valamilyen „nyilvánvaló" módon eleve összeállították.

Végül gondoljuk meg, hogy sok munka végképpen elveszti az aktualitását, ha nem végzik el elég gyorsan. A hatékonyság aligha fontos egy ideiglenes eszköznél, melyet csak néhányszor szándékoznak használni. Ha az eszközök egy rendszere olyan hasznosnak bizonyul, hogy jelentős mennyiségű erőforrást kezd felemészteni, *akkor* gondolhatunk egy hatékonyabb változattal való helyettesítésére. És ebben a helyzetben az a nagy előnyük van, hogy olyan programot kell írnunk, melynek specifikációja pontos, és amely már hasznosnak bizonyult. Ez a programozási tevékenység sikerét biztosító legjobb ismert módszer.

Gyakorlat

3.10 Határozzuk meg, hogy operációs rendszerünkben van-e lehetőség olyan program létrehozására, amelyik beolvas egy csatlakozást leíró sort, és létrehozza a művelet végrehajtásához szükséges operációs rendszerbeli parancsokat. Ha lehet, készítsük el! Ha nem lehet, milyen szolgáltatások hiányoznak és hogy tudnánk ezeket legegyszerűbben pótolni? □

3.7 FILE-OK DINAMIKUS LÉTREHOZÁSA

Olyan területhez érkeztünk, amely a környezettel kölcsönhatásban levő programok szempontjából nagyon fontos. Ez file-ok vagy információs csatornák dinamikus — azaz a program futása közbeni — létrehozása. A problémákat a **makecopy** illusztrálja:

makecopy f g

átmásolja az **f** file-t **g**-be. Ha szükséges, **g**-t létrehozza; ha **g** már létezik, annak tartalma felülíródik.

Hogyan hozzuk létre az outputfile-t? Mivel minden operációs rendszer más szintaxis szerint hajtja végre ezt a műveletet, feltételezzük, hogy a file létrehozását a **create** elemi függvény végzi, melyet gépünkön a megfelelő formában kell majd létrehoznunk. Használata a következő:

belső név **create** *(külső név, elérési mód)*

A **create** és az **open** nagyon hasonlóak: a külső név az a név, amelyet a file a külvilágban visel; a belső nevet itt is a **getlin** és a **putlin** használja. Mint az **open** esetében, a **create** is igényelhet további információt, mint pl. a hozzáféréshez szükséges engedélyek; ezeket az *elérési mód*-ban egyesítettük. Már létező file esetében a **create** először meg kell hogy szüntesse a régi változatot vagy annak hosszát nullára kell hogy állítsa; ez biztosítja azt, hogy egy file újrafelhasználása nem különleges eset. Ha a file létrehozása bármilyen okból is sikertelen, a **create ERR**-t ad vissza.

```
# makecopy — file átmásolása egy másik file-ba
    character iname(NAMESIZE), oname(NAMESIZE)
    integer create, getarg, open
    integer fin, fout

    if (getarg(1, iname, NAMESIZE) == EOF
      | getarg(2, oname, NAMESIZE) == EOF)
        call error("hívás: makecopy bemenet kimenet.")
    fin = open(iname, READ)
    if (fin == ERROR)
        call cant(iname)
    fout = create(oname, WRITE)
    if (fout == ERROR)
        call cant(oname)
    call fcopy(fin, fout)
    call close(fin)
    call close(fout)
    stop
    end
```

Gyakorlat

3.11 Sok operációs rendszer tartalmaz egy „másolás" parancsot, amely hasonló a **makecopy**-hoz. Néha, ha a fogadó file már létezik, a parancs vagy nem hajtódik végre, vagy megerősítést igényel a régi tartalom megsemmisítése előtt. Kívánatos viselkedés-e ez? Mi történjen a következő esetben?

makecopy f f

□

3.8 MINDEZEK EGYÜTT: ARCHÍVUM

Jelen fejezet utolsó példájaként írjunk egy olyan programot, amelyben fölhasználjuk a file-rendszer eddig tárgyalt összes komponensét és még néhány továbbit, melyeket eddig nem említettünk. Az **archive** egy könyvtárkezelő, melynek célja, hogy tetszőleges file-ok halmazait egyetlen nagy file-ba gyűjtse össze, és elvégezze ezen file-nak „archívum"-ként való karbantartását. Ezáltal gyakran tárolóhelyet takarítunk meg, és ami még fontosabb, lehetőséget kapunk arra, hogy összetartozó file-ok egy csoportját tudjuk egyszerre kezelni. Az archívumból file-okat lehet kiválasztani, újabbakat beilleszteni, régieket törölni vagy felújított változatokkal helyettesíteni, és file-ok tartalmát listázni. Ilyen módon az archívum más programok, mint pl. betöltőprogramok, fordítóprogramok stb. számára nyújthat könyvtári szolgáltatásokat. Az **archive** hívása a következő:

archive *parancs archívumnév nem kötelező file-nevek*.

A *parancs* egyetlen betű, amely megmondja, hogy milyen műveletet akarunk végrehajtani az archívumnévvel jelölt archívumon. Az esetleges file-nevek azokra a file-nevekre utalnak, melyek részt vesznek a műveletben. A parancsok a következők lehetnek:

d az archívum adott tagjainak törlése
p tagok nyomtatása a standard outputon
t az archívum tartalomjegyzékének kinyomtatása
u az archívum adott tagjának felújítása vagy új tag csatlakoztatása a végéhez
x adott file-ok kiemelése az archívumból

Az **archive** könnyebb használatának érdekében bevezetjük azt a szabályt, hogy ha egyetlen file sincs megnevezve, akkor a tevékenységet az archívum *összes* file-ján végrehajtjuk:

archive t arch

pl. kilistázza a teljes tartalomjegyzéket. Ellenben, ha bármelyik file-t megnevezzük, akkor csak a megnevezett(ek) vesznek részt a műveletben. Például

archive t arch f g

csak az **f**-re és a **g**-re vonatkozó információt listázza ki. Ezen kívül az **archive** figyelmeztetést ad minden olyan esetben, amikor a megnevezett file nem található az archívumban. Ezeknek a szolgáltatásoknak a megvalósítása nem kerül sokba, de sokat javítanak a program emberi szemléletmódján.

Az archívumkezelő program a „kályhától való elindulás" tipikus esete. Az alapötlet az, hogy kiemeljük a programnak egy kis, kezelhető darabját — egy részletét, ami valami hasznosat csinál — és ezt működőképes állapotba hozzuk. Ha ez már megvan, akkor egyre több és több részt lehet hozzátenni, egészen addig, amíg az egész feladatot meg nem oldottuk. Ha a kezdeti tervezéskor gondosan jártunk el, a későbbi részek viszonylag pontosan illeszkednek. A hibakeresés és a tesztelés köny-

nyebb, mivel a részeket egyenként illesztjük a programhoz. És természetesen ha valamikor bármely okból úgy döntenénk, hogy kidobjuk az egészet, csak az addig elkészült részt csináltuk hiába.

A fenti módszerben az a szép, hogy a program már nagyon hamar elvégzi feladatának egy részét. Azáltal, hogy a leghasznosabb tevékenységeket valósítjuk meg először, kialakul egy benyomásunk a program hasznosságáról, még mielőtt időt fektetnénk a bonyolult vagy csak beavatottaknak szóló szolgáltatások megvalósításába (melyek gyakran még feleslegesnek vagy nemkívánatosnak is bizonyulnak). Ily módon biztosítjuk az egyszerűbb tevékenységek egyszerűbb kezelését is, ami végül a hatékonyságot növeli.

Az első tevékenység, amivel foglalkozunk, a file-ok bevitele az archívumba. (Egy új archívum létrehozása úgy történik, hogy létrehozunk egy üres archívumot, majd file-okat adunk hozzá.) Ameddig nem tudunk archívumot létrehozni és bővíteni, a többi művelet úgysem nagyon érdekes. Így kézenfekvő az a kérdés, hogy milyen formájú legyen egy archívumbeli file. Legalább két lehetőség van. Az egyik, hogy az archívum elején egy katalógusunk van, amelyik felsorolja az archívum file-jait és a rájuk vonatkozó további hasznos információt — hol vannak a file-ok az archívumban, milyen nagyok, mikor lettek archiválva, és így tovább. A másik módszer, amit mi fogunk használni, ennek a katalógus információnak az archívumban való szétosztása a file-ok között. Mindkét megközelítésnek megvannak a maga előnyei és a hátrányai; a 3.15 gyakorlat éppen ezek részletes összehasonlításával foglalkozik. A továbbiak elolvasása előtt mindenki meggondolhatja, hogy az egyes módszerekkel mit lehet majd könnyen és mit nehezen megoldani.

Mint mindig, az adott környezet nagymértékben befolyásolhatja a két módszer megítélését azáltal, hogy egyes műveleteket támogat, másokat pedig akadályoz. Továbbá, bármilyen következtetést vonunk is le, figyelembe kell vennünk, hogy hogyan fogják a programot ténylegesen használni, ami viszont gyakran attól függ, hogy a program milyen tevékenységeket végez el jól, ez pedig megint csak a szervezésen múlik és így tovább.

Központi katalógus esetén az olyan műveletek, mint a tartalomjegyzék kinyomtatása, valószínűleg gyorsabbak lesznek, mivel az információ koncentráltan áll rendelkezésre. Ez jobb hibaellenőrzési lehetőséget is nyújthat, mivel minden információhoz egy időben férhetünk hozzá. Azok a feladatok is egyszerűsödnek, melyeknél az adatokat tárolási sorrendjüktől különböző sorrendben kell elérni. A mérleg másik serpenyőjébe kerül ellenben az a tény, hogy fel kell készülnünk a teljes katalógus együttes kezelésére, ami az archívum tagjainak számát korlátozhatja. Osztott katalógus esetén a tartalomjegyzéken végzett műveletek minden valószínűség szerint lassúbbak lesznek, hacsak rendszerünk nem teszi lehetővé, hogy gyorsan eljussunk egy file tetszőleges pontjára, a közbeeső adatok olvasása nélkül. Még ezen lehetőség mellett is, egy adott file eléréséhez szükséges idő hosszabb lesz, mint központi katalógus esetén, mivel legalább a katalógusbeli bejegyzéseket meg kell vizsgálnunk, amíg a keresettet meg nem találjuk. Ez a lassulás azonban nem jelentős egy file tényleges feldolgozásának idejéhez képest*.

Némi vita és egy központi katalógust használó változat elkészítése után úgy döntöttünk, hogy az **archive**-t osztott katalógussal írjuk meg. Utólag ez tisztább és kevésbé bonyolult szervezésnek tűnik és a hatékonyságveszteség nem jelentős.

Mivel osztott katalógust használunk, az archívum minden tagja egy fejrésszel (headerrel) kezdődik, amely mindenképpen tartalmazza a file nevét és még valami

* A gyakorlatban az osztott katalógust a soros elérésű adathordozók (mágnesszalag), a központi katalógust pedig a véletlen elérésű adathordozók (mágneslemez) esetén célszerű használni *(A lektor)*.

olyan információt, ami lehetővé teszi a fejrésznek és az archívumbeli file tartalmának minden kétséget kizáró módon való megkülönböztetését. Az archívum szerkezete a következő:

file1 fejrésze
file1
file2 fejrésze
file2
...

Ebből a szerkezetből azonnal következik, hogy hogyan kell bizonyos műveleteket megvalósítani. Például a file-nevek kilistázásához csupán a fejrészeket kell megtalálnunk, és ezek megfelelő részeit kinyomtatni. Más műveletek nehezebbek, és attól függenek, hogy milyen szolgáltatásokat nyújt az adott operációs rendszer.

Az **archive** legfelső szintje egy többszörös elágazás, amely az egy-egy parancsnak megfelelő rutint hívja. Bemutatjuk az egészet, bár a kezdeti állapotban csak a pillanatnyilag megírt rutinokat kell hívnia.

```
# archive — file karbantartó
    character aname(NAMESIZE)
    integer getarg
    integer comand(2)

    if (getarg(1, comand, 2) == EOF
      | getarg(2, aname, NAMESIZE) == EOF)
        call help
    call getfns
    if (comand(1) == UPD)
        call update(aname)
    else if (comand(1) == TBL)
        call table(aname)
    else if (comand(1) == EXTR | comand(1) == PRINT)
        call extrac(aname, comand(1))
    else if (comand(1) == DEL)
        call delete(aname)
    else
        call help
    stop
    end
```

Az archívumból való kiemelést és a nyomtatást egyetlen rutinban egyesítettük, mivel csak abban különböznek egymástól, hogy melyik file-ba kell az outputokat küldeni. Úgy döntöttünk, hogy nem használjuk a könnyebben olvasható **UPDATE, TABLE** stb. szimbolikus konstansokat a parancsok jelölésére, bár ezek természetesebb hangzásúak lennének. Sok számítógép azonban csak egyféle betűtípust enged meg, és így ezeket nem lehetne megkülönböztetni az **update**-től és a **table**-től. Törekszünk arra, hogy a betűtípusokból eredő ellentmondásokat elkerüljük programjainkban, hogy ezáltal azok változatlanul átvihetők legyenek ilyen számítógépekre is.

A **help**-et akkor hívjuk, ha az **archive**-ot hibásan használták. Egy könnyen leírható program esetében a leghasznosabb hibajelzés a program használatának tömör leírása.

```
# help — hibaüzenetek kiírása
    subroutine help

    call error("hívás: archive{dptux}név[file-ok]")
    return
    end
```

Ez az üzenet általában elég ahhoz, hogy emlékeztesse a felhasználót, mit kell tennie.

A **getfns** kiemeli a file-név paramétereket a parancssorból és az **fname** tömbben gyűjti össze őket; a file-név paraméterek számát az **nfiles** tartalmazza. A **getfns** egyúttal ellenőrzi, hogy nincs-e ismétlés a paraméterlistában és az nem haladja-e meg a megengedett maximális hosszat.

```
# getfns — file-nevek átvétele fname-ba, ismétlődésvizsgálat
    subroutine getfns
    integer equal, getarg
    integer i, j
    include carch

    errcnt = 0
    for (i = 1; i <= MAXFILES; i = i + 1)
        if (getarg(i+2, fname(1, i), NAMESIZE) == EOF)
            break
    nfiles = i — 1
    if (i > MAXFILES)
        if (getarg(i+2, j, 1) ¬= EOF)
            call error("túl sok file-név")
    for (i = 1; i <= nfiles; i = i + 1)
        fstat(i) = NO
    for (i = 1; i < nfiles; i = i + 1)
        for (j = i + 1; j <= nfiles; j = j + 1)
            if (equal(fname(1, i), fname(1, j)) == YES) {
                call putlin(fname(1, i), ERROUT)
                call error(": ismétlődő file-név")
                }
    return
    end
```

Mivel a FORTRAN a kétdimenziós tömböket oszlop, és nem sor szerint tárolja, az *i*-edik file-névre **fname (1, i)**-vel és nem **fname (i, 1)**-gyel hivatkozunk, így egy név karakterei folytatólagosak lesznek.

Egy megjegyzés PL/1 programozók számára: **fname (1, i)** egy egydimenziós tömb, ebben az esetben az **fname** kétdimenziós tömb egy szelete. A FORTRAN ilyen fogalmakat nem ismer, ezért kénytelenek vagyunk **fname (1, i)**-t írni, ami egyetlen elemnek látszik, pedig mi valójában az egész *i*-edik oszlopot értjük rajta. Ezért PL/1-ben a **getfns** összes **fname (1, i)** alakú hivatkozásait **fname (x, i)**-re kell változtatni. Ez olyan nyelvi probléma, melyet nem lehet mindig elkerülni (bár megpróbáljuk), így mindent elkövetünk annak érdekében, hogy előre felhívjuk rá a figyelmet.

Az **archive** bizonyos esetekben kinyomat egy üzenetet azokról a megnevezett file-okról, amelyek nem szerepeltek az archívumban. Ezt az információt az **fstat**-ban tároljuk; ha **fstat(i)** értéke **NO**, akkor az *i*-edik file még nem szerepelt az archívumban. Ezeket a változókat több rutin igényli, ezért a **carch** nevű közös adatmezőben tartjuk őket, melyet egy **include** utasítással építünk be oda, ahová szükséges.

```
common /carch/ fname(NAMESIZE, MAXFILES), fstat(MAXFILES),
               nfiles, errcnt
   character fname      # file-név paraméterek
   integer fstat        # megnevezett file-ok előfordulnak-e
                        # (YES/NO), kezdetben NO
   integer nfiles       # file-név argumentumok száma
   integer errcnt       # hibaszámláló; kezdetben 0
```

Az **errcnt**-t a hibák számlálására használjuk; több rutin használja, ezért a legjobb szintén a közös adatmezőben tartani.

Figyeljük meg, hogy az **archive** főrutinja nem tud erről a közös adatmezőről, mivel nincs rá szüksége. A FORTRAN **common**-ja kényelmes módot nyújt összetartozó változók csoportjának kezelésére, de ugyanakkor veszélyes is, mivel kevésbé kötött, mint az argumentumok átadása és megnehezítheti annak eldöntését, hogy mely rutinok milyen változókat használnak. A FORTRAN-nál hatásosabb adatszerkezeti lehetőségekkel rendelkező nyelvekben az összetartozó változóknak egy struktúrában való elhelyezése a szokásos megoldás, amit egyetlen argumentumként lehet az őt igénylő rutinoknak átadni. Nem zárjuk ki a FORTRAN **common** vagy a PL/1 **external** változóinak használatát, de ügyelünk arra, hogy, amennyire csak lehetséges, azokra a rutinokra korlátozzuk, melyeknél tényleg szükséges. (Egyes FORTRAN-változatokban **carch**-nak mindenképpen szerepelnie kell a főprogramban.)

A

```
call putlin (fname (1, i), ERROUT)
```

sorban említjük először az **ERROUT** file-t, melyet hibajelzésekre használunk. Feltesszük, hogy ez a file is, az **STDIN** és az **STDOUT**-hoz hasonlóan automatikusan kijelölődik a program indulásakor. Definíció szerint az **error** és a **remark** az **ERROUT**-ba írnak. Bár **ERROUT** lehet a **STDOUT** szinonimája, általában különbözőknek kell lenniük; ily módon az információt adó, de a nem katasztrofális hibákra utaló üzenetek anélkül jelenhetnek meg, hogy elrontanák a fő kimenetet vagy belekerülnének egy csatlakozó programba.

Az archívum felújítása két, jól elkülöníthető szakaszra oszlik: meglevő tagoknak új változatokkal való helyettesítésére és paraméterként megnevezett, de az archívumban nem szereplő file-oknak az archívum végéhez való hozzáillesztésére. Feltesszük, hogy egy file végéhez csak úgy lehet adatokat hozzáilleszteni, hogy átmásoljuk a meglevő információt egy új file-ba, ennek a végéhez beillesztjük az új adatokat és visszamásoljuk az egész file-t az eredetibe. Bár néhány rendszer megengedi a file végéhez való hozzáírást vagy a file belsejének átírását, nem célszerű így eljárni. Biztonságosabb, ha nem változtatjuk meg a létező archívumot addig, amíg nem bizonyosodtunk meg arról, hogy a bekerülő új tag teljes és hibátlan.

A felújítás folyamata a következőképpen foglalható össze:

```
    az archívum megnyitása (létrehozása, ha új)
    ideiglenes file létrehozása
    a meglevő archívum tartalmának felújítása az ideiglenes file-ba
minden új file-ra {
    fejrész létrehozása és beírása az ideiglenes file-ba
    a file másolása az ideiglenes file-ba
    }
ha nincs hiba
    az ideiglenes file visszavitele az archívumba
```

Ezeket a műveleteket az **update** vezérli.

```
# update — meglevő file-ok felújítása, újak felvétele
    subroutine update(aname)
    character aname(NAMESIZE)
    integer create, getarg, open
    integer afd, i, tfd
    include carch
    string tname "archtemp"

    afd = open(aname, READWRITE)
    if (afd == ERR)    # lehet, hogy új file
        afd = create(aname, READWRITE)
    if (afd == ERR)
        call cant(aname)
    tfd = create(tname, READWRITE)
    if (tfd == ERR)
        call cant(tname)
    call replac(afd, tfd, UPD, errcnt)        # a meglevő felújítása
    for (i = 1; i <= nfiles; i = i + 1)       # az új felvétele
        if (fstat(i) == NO) {
            call addfil(fname(1, i), tfd, errcnt)
            fstat(i) = YES
            }
    call close(afd)
    call close(tfd)
    if (errcnt == 0)
        call amove(tname, aname)
    else
        call remark("súlyos hibák — archívum nem módosult")
    call remove(tname)
    return
    end
```

Az **update** először megpróbálja megnyitni az archívumot. Ha nem sikerül, akkor megpróbálja létrehozni, mivel azt feltételezi, hogy új archívumról van szó. Ha egyik művelet sem sikerül, feladatát nem tudja elvégezni.

Az **archive**-t úgy terveztük, hogy egy futáson belül a lehető legtöbb hibát kimutassa. Az **update** feldolgozza a paraméterlistán levő összes file-t, még akkor is, ha valamelyik file megnyitásakor hibát észlel. A hibákat az **errcnt** számolja, és az

archívum csak akkor kerül módosításra, ha a futás végén értéke nulla. Az archívumot **replac** másolja az ideiglenes file-ba és eközben felújítja a specifikált file-okat. Az **addfil** egy file-t illeszt, amennyiben ez lehetséges, az ideiglenes file végéhez. (A **replac**-hoz és az **addfil**-hez rövidesen visszatérünk.) A **remove** a **create** inverze egy file végleges eltávolítását végzi.

Az **amove** az ideiglenes file-ból visszaírja az információt az archívumbeli file-ba, ha nem volt hibajelzés. Legrosszabb esetben ez megoldható a régi archívum fizikai eltávolításával, egy új létrehozásával és azután az ideiglenes file-nak ide való átmásolásával a következő módon:

```
# amove — átvitel name1-ből name2-be
   subroutine amove(name1, name2)
   character name1(ARB), name2(ARB)
   integer create, open
   integer fd1, fd2

   fd1 = open(name1, READ)
   if (fd1 == ERR)
        call cant(name1)
   fd2 = create(name2 WRITE)
   if (fd2 == ERR)
        call cant(name2)
   call fcopy(fd1, fd2)
   return
   end
```

Egyes környezetekben elképzelhető, hogy az ideiglenes file-t egyszerűen átnevezhetjük az új archívummá. Ez nagyon hasznos lehetőség, mivel nyilvánvalóan hatékonyabb, mint a teljes file-nak az átmásolása, és minimumra csökkenti azt az időtartamot, amely alatt az archívum állandó változata befejezetlen állapotban van. A továbbiakban az „átvitel” kifejezést használjuk minden olyan file-mozgatásra, mely átnevezéssel is végrehajtható, és a „másolás” kifejezést használjuk arra az esetre, amikor ragaszkodunk a változatlan forrásinformáció megőrzéséhez.

Következőként foglalkozzunk **addfil**-lel, mivel ez viszonylag önálló.

```
# addfil — a "name" file felvétele az archívumba
   subroutine addfil(name, fd, errcnt)
   character head(MAXLINE), name(ARB)
   integer open
   integer errcnt, fd, nfd

   nfd = open(name, READ)
   if (nfd == ERR) {
        call putlin(name, ERROUT)
        call remark(": a felvétel lehetetlen")
        errcnt = errcnt + 1
        }
   if (errcnt == 0) {
        call makhdr(name, head)
```

```
        call putlin(head, fd)
        call fcopy(nfd, fd)
        call close(nfd)
        }
    return
    end
```

Az egyértelműen megkülönböztethető, és valamennyi archívumbeli file előtt megtalálható fejrész-rekordot a **makhdr** készíti. A próbák idejére bármely kinyomtatható jelsorozat megfelelő. Mi a **—h—** jelzést használjuk, melyet egy szóköz és a file neve követ. „Éles" felhasználás esetén ezt még valószínűleg ki kell egészítenünk a dátummal és a file archiválásának időpontjával.

A további tartalom egy-egy adott operációs rendszer függvénye. Az alapvető célkitűzés annak a lehetőségnek a kiküszöbölése, hogy a fejrész és az archívumbeli file-ok tartalma összekeveredjék. Ha egy file mérete valamilyen alkalmas egységben, mint például a karakterek vagy rekordok száma könnyen és gyorsan kifejezhető, akkor a fejrész tartalmazhatja a file méretét, és nem okoz nehézséget annak eldöntése, hogy mi tartozik fejrészhez, és mi az archívumhoz. A mi **makhdr** változatunk egy karakterszámlálót tartalmaz a fejrészben, amelyet a file nevétől egy szóköz választ el.

```
# makhdr — fejrészkészítés az archívum tagjaihoz
    subroutine makhdr(name, head)
    character head(MAXLINE), name(NAMESIZE)
    integer fsize, itoc, length
    integer i
    string hdr "—h—"

    call scopy(hdr, 1, head, 1)
    i = length(hdr) + 1
    head(i) = BLANK
    call scopy(name, 1, head, i + 1)
    i = length(head) + 1
    head(i) = BLANK
    i = i + 1 + itoc(fsize(name), head(i+1), MAXCHARS)
    head(i) = NEWLINE
    head(i+1) = EOS
    return
    end
```

A **makhdr** felhasználja **length**-t és **itoc**-t, melyeket a 2. fejezetben készítettünk. Az **scopy** egy alapvető jelsorozat-másoló rutin:

```
scopy(from, i, to, j)
```

átmásolja **from**-ból az **i**-től kezdődő karaktersorozatot **to**-ba, **j**-től kezdve.

```
# scopy — karaktersorozat-másolás — from(i) → to(j)
    subroutine scopy(from, i, to, j)
    character from(ARB), to(ARB)
    integer i, j, k1, k2
```

```
    k2 = j
    for (k1 = i; from(k1) ¬= EOS; k1 = k1 + 1) {
        to(k2) = from(k1)
        k2 = k2 + 1
        }
    to(k2) = EOS
    return
    end
```

A **fsize** függvény egy file méretét adja karakterekben. Ideális esetben ez a helybeli file-rendszer egy szolgáltatása. Kevésbé kedvező (de eléggé általános) eset az, hogy egy file méretét csak úgy tudjuk meghatározni, ha végigolvassuk. Bár ez költséges, de ha gyakrabban van szükség az információ visszanyerésére, mint annak kicserélésére, akkor az ismételt olvasással járó külön többletidő is elviselhető. Ne felejtsük el, hogy az **archive** felújításkor kétszer végigolvassa és végigírja a *teljes* archívumot, ezért néhány tagjának eggyel többször való olvasása már nem jelent lényeges többletköltséget.

```
# fsize — file mérete karakterekben
    integer function fsize(name)
    character getch
    character c, name(ARB)
    integer open
    integer fd

    fd = open(name, READ)
    if (fd == ERR)
        fsize = ERR
    else {
        for (fsize = 0; getch(c, fd) ¬= EOF; fsize = fsize + 1)
            ;
        call close(fd)
        }
    return
    end
```

Az **fsize** egyenként olvassa a karaktereket a file-ból a **getch** segítségével, amely a **getc** megfelelője attól eltekintve, hogy egy adott file-t használ a **STDIN** standard input helyett. Ily módon **getc(c)** azonos **getch (c, STDIN)**-nel és **getc** megvalósításának egyik lehetősége a **getch** hívása lehet.

Gyakorlatok

3.12 Hogyan próbálná ki az **archive** eddig elkészült részét? Keressünk néhány kritikus határesetet! □

3.13 Az **fsize** újra megnyitja a file-t, még akkor is, ha ezt **addfil** már megtette. Működik-e ez az Olvasó rendszerében? Kellene-e működnie? Ha nem működik, hogy írná át **fsize-t,** hogy megkerülje ezt a nehézséget? Milyen elemi műveletekre van szükség? □

3.14 Az **update** szubrutin megkísérel egy file-t átmásolni, amelyik létezik, de még semmit nem írtak bele. Milyen viselkedés ésszerű ebben az esetben? Mi történik az Olvasó rendszerében? Milyen elemi művelet tudná ezt a helyzetet kezelni, ha a rendszer ésszerűtlenül viselkedne? □

3.9 TOVÁBBI ARCHÍVUMKEZELŐ PARANCSOK

Most, hogy létre tudunk hozni egy archívumot és file-okat tudunk benne elhelyezni (és feltehetően ezt gondosan ki is próbáltuk), ki tudjuk-e nyomtatni a tartalomjegyzéket? Ez tűnik a következő legegyszerűbb szolgáltatásnak a fokozatos megközelítés elvének értelmében. Emlékezzünk vissza, hogy az **archive** kilistázza a paraméterként adott file-okat, illetve az összes file-t, ha nem kapott paramétereket.

A tartalomjegyzék-nyomtatás művelete alapjában véve a következő:

```
az archívum megnyitása
az archívum minden file-jára {
     if (a fejrész megegyezik valamelyik paraméterrel)
        a fejrész-információ kinyomtatása
        az archívumbeli file átlépése
        }
figyelmeztetés kiírása az összes meg nem talált file-ról
```

A tartalomjegyzék nyomtatását a **table** végzi el:

```
# table — archívum tartalomjegyzékének kiírása
     subroutine table(aname)
     character aname(NAMESIZE), in(MAXLINE), lname(NAMESIZE)
     integer filarg, gethdr, open
     integer afd, size

     afd = open(aname, READ)
     if (afd == ERR)
        call cant(aname)
     while (gethdr(afd, in, lname, size) ¬= EOF) {
        if (filarg(lname) == YES)
              call tprint(in)
        call fskip(afd, size)
        }
     call notfnd
     return
     end
```

A **table** az archívumot csak olvasásra nyitja meg, ily módon olyan archívumot is olvasni tudunk, melynek megváltoztatására esetleg nincsen engedélyünk.

A **tprint** kinyomtatja a fejrészből a kívánt információt; a mi jelen változatunk kinyomtatja a teljes fejrészt, mivel a program tesztelésekor pontosan erre van szükség.

```
# tprint — file fejrészének kiírása
     subroutine tprint(buf)
     character buf(ARB)

     call putlin(buf, STDOUT)
     return
     end
```

A **gethdr** megvizsgálja, hogy fejrész következik-e. Ha igen, visszaadja a fejrészt, a file nevét és méretét. Ha nem talál azonnal egy archívumbeli fejrészt, akkor valami kisiklás történt: a kérdéses file nem archívum vagy tartalma elromlott. Az **archive** egyik esetben sem tudja munkáját folytatni, ezért **gethdr** kilép egy hibaüzenettel.

```
# gethdr — a fejrész tartalmának átvétele fd-ből
    integer function gethdr(fd, buf, name, size)
    character buf(MAXLINE), c,name(NAMESIZE), temp(NAMESIZE)
    integer ctoi, equal, getlin, getwrd
    integer fd, i, len, size
    string hdr "—h—"

    if (getlin(buf, fd) == EOF) {
        gethdr = EOF
        return
        }
    i = 1
    len = getwrd(buf, i, temp)
    if (equal(temp hdr)= =NO)
        call error("az archívum nem szabályos szerkezetű")
    gethdr = YES
    len = getwrd(buf, i, name)
    size = ctoi(buf, i)
    return
    end
```

A **gethdr** az **include** számára készült **getwrd**-t, valamint a **compare** számára írt **equal**-t használja, és nem vezet be új rutinokat ugyanarra a feladatra. A karaktersorozat formájában kapott méret számmá való átalakítását a **ctoi**-vel végezteti el. Kezd kialakulni egy kicsiny segédprogramokból álló hasznos könyvtárunk.

A **makhdr** és **gethdr** nem függ a választott fejrészt bevezető karaktersorozattól; ha nem a **—h—** jelzést akarjuk használni, csupán a **hdr** karakterláncot kell megváltoztatnunk.

Az **fskip** a **gethdr** által adott hosszat (**size**) használja az archívumbeli file átlépésére; ha minden rendben van, a file ezután egy új fejrészhez van pozicionálva a **gethdr** következő hívását várva. Egy kényelmesen megalkotott rendszerben **fskip** elemi művelet, amely olvasás nélkül lépi át a file-t. Kevésbé barátságos környezetben jó a következő változat, amely a megfelelő számú jelet végigolvassa:

```
# fskip — n karakter átlépése az fd file-ban
    subroutine fskip(fd, n)
    character getch
    character c
    integer fd, i, n

    for (i = 1; i <= n; i = i + 1)
        if (getch(c, fd) == EOF)
            break
    return
    end
```

A **filarg** megvizsgálja, hogy valamelyik archívumbeli file-név megtalálható-e a program paraméterlistájában szereplő file-nevek között; az összehasonlításra az **equal**-t használja. Ha nem volt egyetlen file-név paraméter sem megadva, **filarg** ezt is egyezésnek tekinti.

```
# filarg — "name" keresése az argumentum-listában
    integer function filarg(name)
    character name(ARB)
    integer equal
    integer i
    include carch

    if (nfiles <= 0) {
        filarg = YES
        return
        }
    for (i = 1; i <= nfiles; i = i + 1)
        if (equal(name, fname(1, i)) == YES) {
            fstat(i) = YES
            filarg = YES
            return
            }
    filarg = NO
    return
    end
```

Az **fstat(i)**-ben jegyezzük fel, hogy az *i*-edik file-név paraméter szerepel-e az archívumban. Kezdeti értéke minden paraméterre **NO** és a megfelelő pozíciót **YES**-re állítjuk, ha **filarg** egy egyezést talál. Ezt a listát használja **notfnd**, amikor kinyomtatja az archívumban meg nem talált argumentumokat.

```
# notfnd — "nincs meg" üzenet kiírása
    subroutine notfnd
    integer i
    include carch

    for (i=1; i<=nfiles; i = i + 1)
        if (fstat (i)==NO) {
            call putlin(fname(1, i), ERROUT)
            call remark(":nincs az archívumban")
            errcnt = errcnt + 1
            }
    return
    end
```

A **notfnd** a hibaszámlálót is beállítja, de ezt **table** nem használja föl, hiszen nincs jelentősége akkor, ha nem akarjuk az archívumot változtatni.

A fokozatos fejlesztés következő lépése a file-kiemelés, az archívumban már elhelyezett információ visszanyerése. Logikája elég világos:

```
az archívum megnyitása
az archívum minden file-jára
    if (a file kivitelre kerül) {
        egy file létrehozása
        másolás az archívumból a file-ba
        }
figyelmeztetés azokról a file-okról, amelyek kiemelése sikertelen volt.
```

Azt is megengedjük, hogy az archívumbeli file-ok tartalmát a standard outputon gyűjtsük össze, külön megadott file-ok helyett. A **p** parancs lehetővé teszi, hogy kilistázzuk az archívum tagjait, vagy tárolási nevüktől különböző néven emeljük ki őket az archívumból. Például:

```
archive p arch file1 >file2
```

kiemeli **file1**-et és elhelyezi **file2**-ben. Természetesen ez csak akkor használható, ha a program nem ad olyan locsogásszámba menő üzeneteket, mint:

```
sikeres kiemelés
1 file kiemelése megtörtént
file1 tartalma:
```

stb. A fölösleges üzenetek elkerülése részint olyan környezetben kívánatos, ahol a fölösleges megjegyzések megzavarják a programok sima összekapcsolódását, részint mint tervezési elv, kedvére való azoknak, akiknek hamar unalmassá válnak a túl sokat beszélő programok.

```
# extrac — file-ok kiemelése az archívumból
   subroutine extrac(aname, cmd)
   character aname(NAMESIZE), ename(NAMESIZE), in(MAXLINE)
   integer create, filarg, gethdr, open
   integer afd, cmd, efd, size
   include carch

   afd = open(aname, READ)
   if (afd == ERR)
      call cant(aname)
   if (cmd == PRINT)
      efd = STDOUT
   else
      efd = ERR
   while (gethdr(afd, in, ename, size) ¬= EOF)
      if (filarg(ename) == NO)
         call fskip(afd, size)
      else {
         if (efd ¬= STDOUT)
            efd = create(ename, WRITE)
         if (efd == ERR) {
            call putlin(ename, ERROUT)
            call remark(": nem lehet létrehozni")
```

```
                errcnt = errcnt + 1
                call fskip(afd, size)
                }
            else {
                call acopy(afd, efd, size)
                if (efd ¬= STDOUT)
                    call close(efd)
                }
            }
    call notfnd
    return
    end
```

A legbonyolultabb rész most sem maga a művelet, hanem a hibafelderítés és a hibakezelés.

Az **acopy** az archívum egy tagját kimásolja egy file-ba, felhasználva a fejrészben levő hossz-információt ahelyett, hogy a következő fejrészt megkeresné. (Az **EOF** figyelése biztonsági intézkedés.) Ily módon az **acopy** bármilyen információ másolására használható — teljesen független az archivált file-ok tartalmától.

```
# acopy — size számú karakter másolása fdi-ből fdo-ba
    subroutine acopy(fdi, fdo, size)
    character getch
    character c
    integer fdi, fdo, i, size

    for (i = 1; i <= size; i = i + 1) {
        if (getch(c, fdi) == EOF)
            break
        call putch(c, fdo)
        }
    return
    end
```

A **putch** a **getch** inverze; karaktereket ír egy adott file-ba.

A törlés lényegében azonos a felújítással, attól eltekintve, hogy a törölt file helyére nem kerül másik; a **replac** használható mind az **u**, mind a **d** parancsnál. Az

```
# delete — file-ok törlése az archívumból
    subroutine delete(aname)
    character aname(NAMESIZE,) in(MAXLINE)
    integer create, open
    integer afd, tfd
    include carch
    string tname "archtemp"
```

```
    if (nfiles <= 0)    # védelem a tájékozatlanok ellen
        call error("törölni csak név szerint lehet")
    afd = open(aname, READWRITE)
    if (afd == ERR)
        call cant(aname)
    tfd = create(tname, READWRITE)
    if (tfd == ERR)
        call cant(tname)
    call replac(afd, tfd, DEL, errcnt)
    call notfnd
    call close(afd)
    call close(tfd)
    if (errcnt == 0)
        call amove(tname, aname)
    else
        call remark("súlyos hibák — az archívum nem változott")
    call remove(tname)
    return
    end
```

Az

archive d archname

parancs illegális — nem törli az archívum összes file-ját. Ha ezt akarjuk megvalósítani, explicit módon kell az összes file-t eltávolítani. Ez apró finomság, de biztonságosabbá teszi a programot.

Végül, a **replac** a következő:

```
# replac — file-ok cseréje vagy törlése
    subroutine replac(afd, tfd, cmd, errcnt)
    character in(MAXLINE), uname(NAMESIZE)
    integer filarg, gethdr
    integer afd, cmd, errcnt, size, tfd

    while (gethdr(afd, in, uname, size) ¬= EOF)
        if (filarg(uname) == YES) {
            if (cmd == UPD)      # az új felvétele
                call addfil(uname, tfd, errcnt)
            call fskip(afd, size)    # a régi kihagyása
            }
        else {
            call outlin(in, tfd)
            call acopy(afd, tfd, size)
            }
    return
    end
```

A **replac** az **addfil**-t használja egy file új változatának az ideiglenes file-ba való átmásolására, az **fskip** az archivált változat átlépését oldja meg, az **acopy** pedig a változatlanul átkerülő tagokat másolja.

Az eddigiek során föltűnhetett az Olvasónak, hogy az **archive** programjának mintegy fele hibaellenőrzéssel foglalkozik. A programok jelentős része megengedheti magának, hogy elnézőbb legyen a felhasználókkal szemben az operációs rendszertől és saját tapasztalatlanságuktól való védelem tekintetében, mivel a hibás használat valószínűleg nem jár végzetes következményekkel. A hibakezelés azonban különlegesen fontos olyan programoknál, mint az **archive,** mivel az megváltoztat file-okat, és nem egyszerűen újakat hoz létre. Ez a program létező file-okat ír felül feltételezés szerint hibátlan új tartalommal, ezért nagyon óvatosan kell eljárnia. Például, őrültségnek tűnhet feladni a teljes felújítási műveletet csupán azért, mert a file-ok egyikét nem tudjuk elérni. De a biztonság az első! Jobb a futtatást újra elvégezni, mint az archívum megsemmisítését kockáztatni.

3.10 PROGRAMSZERKEZET

Lehet, hogy az Olvasó mostanra zavarba jött. Nem annyira az **archive** bonyolultsága, hanem amiatt, mert az igen sok darabból áll. Sok része azonban már régi ismerősünk. Ilyenek a file-karbantartó elemi műveletek: az **open,** a **create,** a **close** és a **remove;** a beviteli és kiviteli műveletek, valamint a **getlin,** a **getarg** és a **putlin;** a hibakezelők: a **cant,** az **error** és a **remark;** valamint a segédprogramok: **ctoi, itoc, length, scopy, equal** és **getwrd.** Hozzá kell szoknunk, hogy úgy gondoljunk ezekre az alacsony szintű segédeszközökre, mint *nyelvi bővítésekre,* amelyek segítségével az alapvető műveleteket tömören és figyelmünknek a valódi feladattól való elvonása nélkül tudjuk kifejezni.

Ami még a program áttekinthetőségéhez szükséges, az kifejezhető az **archive** szubrutinjainak hierarchikus szerkezetével. Ez megmutatja a program struktúráját azáltal, hogy megjelöli, melyik rutin melyiket hívja feladatának elvégzése során. Könnyű meggyőződni arról, hogy az első tervhez viszonyítva a program hierarchikus szerkezete sokkal kevesebbet változik, mint szövege, ezért a programhierarchia hasznos kiegészítő dokumentációja az aktuális programlistáknak — nem úgy, mint a blokkdiagramok, melyek csupán a program szövegét tükrözik, és hamar elavultakká válnak a program módosításai következtében. Egyesek még modulkapcsolat-ábrákat is rajzolnak, amelyek a hívások hierarchiáját és az átadott, illetve visszakapott paramétereket tüntetik fel minden hívásnál. Az **archive** esetén a hierarchia a következő:

```
archive
    help
    getfns
    update
        replac
            gethdr, filarg, fskip, addfil, acopy
        addfil
            makhdr, fcopy
        amove
    table
        gethdr, filarg, fskip, notfnd, tprint
    extrac
        gethdr, filarg, fskip, notfnd, acopy
    delete
        replac, notfnd, amove
```

Ez azt jelenti, hogy az **archive,** feladata elvégzéséhez meghívja a **help, getfns, update, table, extrac** és **delete** rutinokat (az előbb említett különböző alacsony szintű rutinokon kívül). Az **update** meghívja a **replac, addfil** és **amove** rutinokat. A **replac** és **addfil** még további rutinokat hív meg és így tovább. Egyes csoportokat a függőleges elhelyezés helyett, helykímélés céljából vesszőkkel elválasztva vízszintesen írtunk. Az ugyanazon rutinhoz tartozó hierarchiát csak egyszer tüntettük fel (lásd a **replac**-ot az utolsó sorban), és ugyanebből az okból kihagytuk az összes alacsony szintű rutint. Ez csak bonyolítaná az ábrát. A továbbiakban is be fogjuk mutatni a nagyobb programok hierarchiáját, hogy ezáltal segítséget nyújtsunk a szerkezet áttekintéséhez.

A hierarchia több dolgot is felfed. Először is az **archive** alapvető szerkezeti eleme egy **case** utasítás, amely meghívja az **update, table, extrac** vagy **delete** négy lehetséges eset egyikét. Mindegyik esetet néhány műveleti rutinná fejtjük ki. A **gethdr, filarg** stb. hasonló szerepet játszik, mint az **open, getlin** és a többi elemi rutin — lehetővé teszi, hogy a végrehajtandó alapvető tevékenységek magasabb absztrakciós szinten (és olvashatóan) legyenek kifejezve, és elkülönítsék az **archive**-ot az alacsony szintű, megvalósítástól függő döntésektől. Minden egyes szinten az alacsonyabb szintű rutinokat építőkövekként használjuk, melyek valamilyen feladatot elvégeznek, úgy, hogy használatukhoz viszonylag keveset kell tudni arról, hogy hogyan hajtják azt végre.

Figyeljük meg, hogy egyetlen rutinnak sem kell kb. féltucatnál több közvetlen alárendeltjéről tudnia. A tapasztalat szerint egy ember nem tud egyidejűleg kb. 7 dolognál többet megfelelően áttekinteni, így ez a hierarchia valamilyen mértékben biztosítja azt, hogy a programterv egyetlen része sem lesz túl nehezen felfogható. Végül érdemes megfigyelni, hogy az annyira különböző műveletek, mint az **update,** a **table** és a **delete,** lényegében ugyanazokat a műveleti rutinokat használják különböző sorrendben. A gondos tervezésnek köszönhető, hogy a közös műveletek jól azonosíthatók és alaptevékenységekké alakíthatók. Az ily módon felépített programokat könnyebb belőni és karbantartani, mivel az egyes főágakkal **(update, table** stb.) és az egyes műveletek részleteivel **(gethdr, filarg** stb.) külön-külön tudunk foglalkozni.

Gyakorlatok

3.15 Most, hogy megismertük az **archive** osztott katalógus változatát, tervezzünk és valósítsunk meg egy olyan változatot, melyben a katalógus az archívum elején helyezkedik el. Vessük össze az előnyöket és a hátrányokat! A „hatékonyság”-gal kapcsolatos megfigyeléseket az archívumot valóban használó emberek gyakorlati tapasztalataival kell alátámasztani! □

3.16 Az **archive** egyszerűbb és gyorsabb lehet, ha az operációs rendszer tartalmaz bizonyos elemi műveleteket. Beszéltünk az ideiglenes file átnevezéséről az átmásolása helyett, egy file méretének meghatározásáról annak végigolvasása nélkül, egy file-on belüli pozicionálásról a közbenső információk elolvasása nélkül, valamint egy file részeinek szelektív átírhatóságáról. Vizsgáljuk meg ezeket és más, esetleg hasznos elemi műveleteket! Rendelkezésre állnak-e ezek az Olvasó rendszerében? Rendelkezésre állhatnának-e? Hogyan érintik ezek az elemi műveletek a katalógust elöl tartalmazó változatot? □

3.17 A legtöbb rendszerben egy archiváló program helyet takarít meg, mivel kisebb file-okat egyetlen nagy file-lá gyűjt össze. Ez kiküszöböli azt az elaprózódást, amely abból ered, hogy minden file egész számú blokkot vagy pályát foglal el a megfelelő háttértárban. Elméletileg mennyi lehet ez a megtakarítás? Mérjük

meg és vizsgáljuk meg, hogy az elméleti előrejelzés mennyire egyezik a gyakorlattal? □

3.18 Az **archive** nem ad figyelmeztetést az **archive u arch** paranccsal kapcsolatban. Kellene-e adnia? Ha elérhető egy file utolsó változtatásának dátuma és időpontja, mi lenne ennek az utasításnak egy hasznos értelmezése? Szükség van-e külön parancsra egy új archívum létrehozásához és egy meglevő felújításához? Kell-e figyelmeztetést adni, amikor egy új archívumot hozunk létre? □

3.19 Ha interaktív környezetben dolgozunk, egészítsük ki az **archive**-ot egy „bőbeszédű" üzemmóddal, amely üzeneteket ad futás közben arról, amit csinál, és választást ajánl a felhasználónak minden egyes file-nál, hogy a parancs végrehajtódjék-e vagy sem. □

3.20 Hogyan tudunk egy **f** file-t bevinni egy archívumba, amely már tartalmaz egy **f**-et? Meg kell-e engednünk az archívumban többszörös előfordulásokat? Ha két archívumot a **concat**-tal egyesítünk, helyes archívum lesz-e az eredmény? □

3.21 Az **archive** nem nyújt lehetőséget arra, hogy megadjuk az archívumhoz adandó file-ok pozícióját az archívumon belül. Szükség van-e erre? Milyen szintaxist használnánk? Valósítsuk meg! □

3.22 Archívumbeli file-jaink mindig a lehető legkisebb méretűek, mivel minden változtatáskor újra létrehozzuk őket. Egyes rendszerek a mi **archive**-ünkhöz meglehetősen hasonló, olyan programokat tartalmaznak, amelyek nem szabadítják fel a helyet, hacsak ezt explicit módon nem igényeljük. Vitassuk meg ennek a módszernek a jó és rossz tulajdonságait! □

3.11 ÖSSZEFOGLALÁS

Ez a fejezet azzal foglalkozott, hogy hogyan teremtenek kapcsolatot a programok a környezetükkel, és elsősorban azzal, hogy hogyan jutnak hozzá a file-okban tárolt információhoz. Kevés rendszer szabályos és következetes a kapcsolattartási módszereket illetően, és természetesen nagy eltérések vannak az egyes rendszerek között. Megpróbáltuk az anyagot és a programokat néhány elemi művelet köré csoportosítani, amelyek vagy rendelkezésre állnak, vagy legalábbis nagy valószínűséggel létrehozhatók a legtöbb operációs rendszerben. A rendszerrel való kommunikációnak ilyen elemi műveletekre való korlátozása az általunk ismert legjobb megközelítése annak a problémának, amit az operációs rendszerek elijesztő sokrétűsége okoz. File-rendszerünkben a következő elemi műveleteket használtuk:

open Egy információállomány („file") külső nevét egy, a program által használt belső névhez rendeli. A file kezdőpozícióba kerül. Ez a művelet a legtöbb rendszerben hagyományos módon, vezérlőkártyával valósítható meg. Nagyon hasznos, de ritkábban megengedett egy file-nak a futó programban megadott külső névvel való megnyitása. Célszerű megengedni egy már nyitott file újabb példányának megnyitását.

create Új file, lehetőleg név szerinti létrehozása egy futó programból. Ha a file már létezik, a régi változatát el kell távolítani vagy méretét nullára zsugorítani és felülírni. A **create** igen fontos elemi művelet, de a legkevésbé valószínű, hogy rendelkezésünkre áll. Még a nagy, modern operációs rendszerek sem mindig tartalmazzák.

close Megszakítja az **open**-nel vagy **create**-tel létrehozott hozzárendelést. Ha szükséges, megjelöli a file végét, hogy későbbi olvasáskor megtaláljuk az **EOF**-et.

remove Eltávolít egy file-t a file-rendszerből.

Föltételezzük, hogy amikor az operációs rendszer elindít egy programot, a **STDIN**, **STDOUT** és **ERROUT** file-ok már megnyitottak és felhasználásra készek. Ugyancsak feltesszük, hogy amikor egy program bármilyen okból is befejeződik, minden file rendben lezáródik. További feltételezés szerint a file megfelelő méretét a rendszer alakítja ki; ha több információt viszünk egy file-ba, az automatikusan kiterjesztődik. Megítélésünk szerint nyilvánvaló, hogy egy file-nak olyan nagynak kell lennie, amekkorára szükség van, és hogy növekednie kell, ha több információt írunk bele, de igen sok rendszeren ezt a felhasználónak magának kell megszerveznie, többnyire igen nehézkes módon.

A jó operációs rendszer egyik ismérve, hogy a fenti műveletek mindegyike egyöntetű, könnyen felhasználható és lényeges kivétel nélkül használható módon, bármely programból származó bármely file-ra rendelkezésre áll. Az Olvasó hasznos tapasztalatokra tehet szert, ha elolvassa operációs rendszerének néhány kézikönyvét és ellenőrzi, hogy a rendszer mennyire teljesíti ezeket a kritériumokat.

BIBLIOGRÁFIAI MEGJEGYZÉSEK

A file-összehasonlítás problémáját még nem oldották meg teljesen, de különböző formáinál eredmények születtek M. L. FREDMAN és mások által kidolgozott algoritmusok alapján. Ezek az algoritmusok meghatározzák a változtatásoknak azt a minimális halmazát, amely egy *n*-soros file-nak egy másik file-ra való konvertálásához szükséges és legrosszabb esetben $n^2 \log n$ időt igényelnek. Idejük tipikus esetben sokkal jobb, gyakran a file méretével egyenesen arányos. Az alapeljárás nem nagyon bonyolult, de számottevő segítséget igényel a rendelkezésre álló operációs rendszertől abból a célból, hogy ismeretlen hosszúságú sorokat tudjon kezelni nagy tárpazarlás nélkül. Részletesebben lásd M. D. MCILROY és J. W. HUNT: „An algorithm for differential file comparison", Bell Laboratórium, Computing Science Technical Report 40, 1976.

Bár számottevő mennyiségű irodalom foglalkozik file-rendszerek megvalósításával, ezek viszonylag kevéssé térnek ki a programozók és a rendszer kapcsolatára. A mi B/K elemi műveleteink legtöbbjének a modellje az UNIX operációs rendszer, amely egy különlegesen jól áttekinthető készletet tartalmaz. Lásd: „The UNIX time-sharing system", amelyet K. L. THOMPSON és D. M. RITCHIE írt a CACM 1974. júliusi számában.

A modulkapcsolat-ábrák leírása megtalálható W. P. STEVENS, G. J. MEYERS és L. L. CONSTANTINE: „Structured design" című cikkében, IBM Systems Journal 1974. április.

4. fejezet

Rendezés

A rendezés mindennapos programozási feladat és gyakran alkotja nagyobb folyamatok egy-egy részét. Ebben a fejezetben a rendezéssel foglalkozunk, de nagyobb súlyt helyezünk a rendezőprogram és a felhasználók kapcsolatára, mint valamilyen „lehető legjobb" rendezési algoritmus bemutatására. Ha egy rendezőprogram alkalmazása olyan bonyolult, hogy helyette inkább magunknak írunk programot, akkor a rendezési algoritmus minősége nyilvánvalóan közömbös. Viszont, ha a program könnyen használható, ha a felhasználók a rendezést kényelmes *eszköznek* fogadják el, akkor azt majd alkalmazzák és szükség szerint az algoritmus is tökéletesíthető; azaz mindenki jól jár.

4.1 BUBORÉKOS RENDEZÉS

Minden programozó számára ismerős a felcseréléses rendezés valamilyen változata. Például a *buborékos rendezés* a következőképpen rendez egy tömböt növekvő sorrendbe:

```
# bubble — v(1)...v(n) buborékos rendezése növekvő sorrendbe
  subroutine bubble(v, n)
  integer i, j, k, n, v(n)

  for (i = n; i > 1; i = i - 1)
      for (j = 1; j < i; j = j + 1)
          if (v(j) > v(j+1)) {          # összehasonlítás
              k = v(j)                   # felcserélés
              v(j) = v(j+1)              #
              v(j+1) = k                 #
              }
  return
  end
```

A belső ciklus minden menetben átrendezi a helytelen sorrendben álló szomszédos elemeket; a menet végére a legnagyobb elem, mint folyadékban a buborék, a felszínre, vagyis a tömb végére, azaz **v(i)**-be kerül. A külső ciklus ezt a folyamatot ismétli, minden alkalommal 1-gyel csökkentve az aktuális felső tömbhatárt.

A buborékos rendezés fő előnye az egyszerűség. Komoly hátránya, hogy ha a rendezendő elemek száma túlságosan naggyá válik, akkor az eljárás erősen lelassul. A buborékos (és hasonló) rendezések *időbeli bonyolultsága (komplexitása)* n^2. Ez azt

jelenti, hogy a rendezés által igényelt idő úgy nő, mint a rendezendő elemek számának négyzete: kétszer annyi elem rendezése kb. négyszer annyi ideig tart. Mit nevezünk „túlságosan" nagynak? Ez függ a szóba jöhető alternatív megoldástól és attól, hogy milyen gyakran kell az adatokat rendezni, de kb. 10—50 elem lehet egy olyan ésszerű határ, amely fölött a buborékos rendezést már célszerűbb valamilyen bonyolultabb algoritmussal helyettesíteni. A gyakorlatban a rutint valószínűleg **sort**-nak és nem **bubble**-nak fogjuk hívni, hogy az algoritmus a felhasználók zavarása nélkül változtatható legyen. Mi a **bubble** nevet használjuk, mivel minden programnak egyedi nevet akarunk adni.

4.2 SHELL-RENDEZÉS

Bonyolultság tekintetében a buborékos rendezés után bizonyos értelemben a Shell-rendezés jelenti a következő szintet. Azért mutatjuk be, mert elvileg hasonlít hozzá: tömör, de nagyobb tömbök esetében sokkal gyorsabb. A Shell-rendezés időbeli bonyolultsága $n^{1,5}$.

A Shell-rendezés alapgondolata az, hogy kezdetben az egymástól messze eső elemeket hasonlítjuk össze a szomszédosak helyett. Ez a módszer a nagymértékű rendezetlenséget gyorsan csökkenti és így későbbre kevesebb munka marad. Az összehasonlított elemek távolsága fokozatosan csökken, amíg el nem éri az 1-et, amikor is a rendezési algoritmus átmegy a szomszédos elemek felcserélését alkalmazó módszerbe.

```
# shell — v(1)...v(n) Shell-rendezése növekvő sorrend be
    subroutine shell(v, n)
    integer gap, i, j, jg, k, n, v(n)

    for (gap = n/2; gap > 0; gap = gap/2)
        for (i = gap + 1; i <= n; i = i + 1)
            for (j = i - gap; j > 0; j = j - gap) {
                jg = j + gap
                if (v(j) <= v(jg))          # összehasonlítás
                    break
                k = v(j)                    # felcserélés
                v(j) = v(jg)                #
                v(jg) = k                   #
                }
    return
    end
```

A külső ciklus az összehasonlított elemek közötti távolságot vezérli. Ez kezdetben $n/2$, és minden menetben feleződik mindaddig, amíg zérussá nem válik. A középső ciklus az egymástól a **gap** által elválasztott elemeket hasonlítja össze; a legbelső ciklus pedig megcseréli azokat, amelyek fordított sorrendben vannak. Mivel a **gap** végül 1-re csökken, az elemek helyes sorrendbe rendeződnek.

Néhány szó a modularitásról. Sok rendezési eljárás három különálló részből áll. Egy *összehasonlító művelet* eldönti két elem sorrendjét. Egy *felcserélő művelet* kicserél két helytelen sorrendben levő elemet. Végül a *rendezési algoritmus* határozza meg, hogy milyen összehasonlításokat és cseréket kell végezni. Két rendezési eljárás

gyakran csak ebben az utóbbiban különbözik egymástól, ezért a programot ennek figyelembevételével gondosan kell szervezni. Ha ez a három rész elkülönül egymástól, egyedileg lehet mindegyiket tökéletesíteni a másik kettő érintése nélkül.

Gyakorlatok

4.1 Hogyan tesztelnénk egy rendezőprogramot? Melyek azok a nyilvánvaló határesetek, amelyeket ellenőrizni kell, hogy a hibátlan működés biztosítva legyen? Milyen programokat írnánk a kipróbálás támogatására? Milyen programokat írtunk az eddigiekben, amelyek ebben segítenek? □

4.2 Tapasztalat útján hasonlítsuk össze a buborékos és a Shell-rendezést. Hol van az a pont, ahol a Shell-rendezés jobbá válik gépünkön? □

4.3 A Shell-rendezés általunk adott változatában, amikor két, fordított sorrendben levő elemet megtalálunk, azokat azonnal felcseréljük. Azonban ha egy elem a pillanatnyi intervallumhoz viszonyítva kicsi, előfordulhat, hogy sok felesleges cserét hajtunk végre. Szervezzük át a cseréket úgy, hogy a tömb eleje felé mozgó elemek egy ideiglenes tárolóban várakozzanak, ameddig a nekik megfelelő pozíciót a program meg nem találja. Elemezzük ezt a változatot és döntsük el, hogy a sebességnövekedés megéri-e az algoritmus áttekinthetőségében beállt veszteséget? □

4.4 Megfigyelték, hogy a Shell-rendezés valamivel gyorsabb, ha a **gap** értéke mindig páratlan. Végezzük el az ennek megfelelő módosítást és kísérleti úton vizsgáljuk meg hatását! □

4.3 SZÖVEGEK RENDEZÉSE

Mivel a jelen könyvben tárgyalt eszközök nagy része szövegmanipulációt végez, rendezési eljárásainkat is célszerű ilyen jellegű műveletekhez adaptálni. Különösen hasznos az a program, amely egy szövegfile-t növekvő lexikografikus sorrendbe rendez, soronként. Amint hamarosan meglátjuk, ez a művelet önmagában is és más folyamatok részeként is hasznos.

A tervezés során két alapvető szempontot kell szem előtt tartanunk: a kényelmet és a hatékonyságot. A **sort**-ot rendkívül könnyen kell tudnunk használni, ne legyen szükség semmilyen előkészületre az általános rendezési feladatok esetén. Ugyanakkor elfogadhatóan hatékonynak (azaz olcsónak) is kell lennie mind kicsiny, mind pedig mérsékelten nagy file-ok esetén. A **sort**-ot nem arra szánjuk, hogy üzemszerűen használt programrendszerek megtervezett rendezőprogramjait helyettesítse, de olyan eszköznek kell lennie, amely kényelmes és gazdaságos a széles határok között változó inputfile-méretek esetén, és amely az alkalmi felhasználót arra ösztönzi, hogy természetes módon ezt válassza. Az első vázlat olyan program, amely a tárban egyidejűleg tárolható sorok halmazát rendezi. Később kiterjesztjük olyan file-ok kezelésére, amelyek nagyságuk miatt nem tárolhatók egyszerre a tárban.

A rendezőprogramot úgy írjuk meg, hogy a standard inputról olvasson és a standard outputra írjon, és ily módon szűrőként legyen használható. Természetesen a **sort** addig semmilyen eredményt sem tud előállítani, amíg a teljes bemenetet végig nem olvasta. (Miért?) Ezért szűrőnek nevezni egy kicsit erőltetettnek tűnhet, de ez nem lényeges. A lényeg az, hogy a felhasználó szempontjából a program szűrőként viselkedik, és ezért programcsatlakozásokban használható. Később megmutatjuk, hogy ez a szervezés hogyan teszi a rendezést rugalmas eszközzé.

A fejezet elején bemutatott szubrutinok egész számok tömbjeit rendezték; most viszont szövegsorokat akarunk kezelni. Sajnos, a sorok nem egyforma hosszúak, és

a karakterek száma általában sokkal nagyobb, mint az eddig kezelt egészek jegyeinek száma, ezért nem lehet őket közvetlenül mozgatni vagy összehasonlítani, legalábbis nem FORTRAN-ban. Milyen adatábrázolás lesz alkalmas változó hosszúságú sorok hatékony és kényelmes kezelésére?

Az egyik megoldás, hogy a sorokra *indirekt módon,* mutatókkal hivatkozunk. A **linbuf** nevű tömb tartalmazza a rendező sorokat, teljesen tömör tárolásban. Egy másik tömb, a **linptr,** indexeket tartalmaz, amelyek azt mutatják, hogy a megfelelő sorok a **linbuf**-on belül hol kezdődnek. Vagyis **linptr(i)** az *i*-edik sor kezdetének indexe **linbuf**-on belül. Amikor cserére kerül sor, a mutatókat cseréljük, és nem magukat a szövegsorokat. Ezzel kiküszöböljük azt a kettős problémát, melyet a bonyolult tárkezelés és a szövegsorok mozgatásához szükséges számottevő többletidő jelent.

A tárban elhelyezett adatokon operáló **sort** a következő:

```
# sort — szövegsorok rendezése a tárban
    character linbuf(MAXTEXT)
    integer gtext
    integer linptr(MAXPTR), nlines

    if (gtext(linptr, nlines, linbuf, STDIN) == EOF) {
        call shell(linptr, nlines, linbuf)
        call ptext(linptr, nlines, linbuf, STDOUT)
        }
    else
        call error("nem rendezhető, mert túl nagy")
    stop
    end
```

Itt a **shell** egy módosított változatát hívjuk, amelyet rövidesen bemutatunk, és amelyik a **linptr**-ben a mutatókat mozgatja, nem pedig magukat a sorokat. A **getext** beolvassa a sorokat és beállítja a mutatókat, **ptext** pedig a mutatók felhasználásával viszi ki a rendezett sorokat.

A **gtext EOF**-et ad vissza, ha bevitel közben a file végéhez ér; ezt itt csak hibaellenőrzésre használjuk. A **linptr(i)**-t az *i*-edik sor első karakterének indexével töltjük fel. **MAXLINE** a leghosszabb sor hossza, amelynek kezelésére felkészültünk, **MAXPTR** pedig a sorok maximális száma.

```
# gtext — szövegsorok bevitele linbuf-ba
    integer function gtext(linptr, nlines, linbuf, infile)
    character linbuf(MAXTEXT)
    integer getlin
    integer infile, lbp, len, linptr(MAXPTR), nlines

    nlines = 0
    lbp    = 1
    repeat {
        len = getlin(linbuf(lbp), infile)
        if(len == EOF)
            break
```

```
        nlines = nlines + 1
        linptr(nlines) = lbp
        lbp = lbp + len + 1          # "1" = hely az EOS számára
        } until (lbp >= MAXTEXT—MAXLINE | nlines >=MAXPTR)
    gtext = len
    return
    end

# ptext — szövegsorok kivitele linbuf-ból
    subroutine ptext(linptr, nlines, linbuf, outfil)
    character linbuf(ARB)
    integer i, j, linptr(MAXPTR), nlines, outfil

    for (i = 1; i <= nlines; i = i + 1) {
        j = linptr(i)
        call putlin(linbuf(j), outfil)
        }
    return
    end
```

A programnak PL/1-re vagy más nyelvre való fordítása során problémák merülnek fel amiatt, ahogyan a FORTRAN az aktuális paraméterként átadott tömbelemeket kezeli. A hívott rutinok a **linbuf**-ra való indexelt hivatkozásokat a **linbuf**-on belüli tömbrészek kezdőpontjaiként, nem pedig egyetlen tömbelemként használják. A PL/1-ben a tár és a mutatók kijelölésére jobb módszerek is vannak, de ha csupán a fenti programot akarjuk PL/1-re átültetni, akkor csak kevés változtatásra van szükségünk. Először is egy karakterekből álló tömböt deklarálunk (a **line**-t), aztán megváltoztatjuk a **linbuf**-ra való indexelt hivatkozásokat **linbuf(i)**-ről **addr(linbuf(i))→→line**-ra. Itt a PL/1 bőbeszédűbbnek tűnhet, de elkerüli azt a FORTRAN-ban felmerülő problémát, hogy egy tömbre való hivatkozás formailag azonos egy skalárra való hivatkozással.

Természetesen **gtext**-et és **ptext**-et megírhatjuk és kipróbálhatjuk rendezési módszerünktől függetlenül is. Sőt, rendezésre nincs is szükség: egy üres **shell**, amely csupán visszaadja a vezérlést anélkül, hogy valamit is csinálna, elégséges, hogy meggyőződhessünk arról, hogy **gtext** tényleg olyan tömböket hoz-e létre, amelyeket **ptext** megfelelő módon értelmez. Ez a fokozatos fejlesztés és tesztelés tipikus példája. Könnyebb egy olyan programot tesztelni, melynek részei különálló tevékenységet végeznek és csupán áttekinthető, jól definiált módon kapcsolódnak egymáshoz, mint egyetlen rutin szövevényes kódjával birkózni.

Rendezőprogramunkat azért is módosítanunk kell, hogy az összehasonlítást és a cserét szubrutinként hajtsa végre. A Shell-rendezés módosított változata a következő:

```
# shell — Shell-rendezés szövegsorokra
    subroutine shell(linptr, nlines, linbuf)
    character linbuf(ARB)
    integer compar
    integer gap, i, ig, j, k, linptr(ARB), nlines

    for (gap = nlines/2; gap > 0; gap = gap/2)
        for (j = gap + 1; j <= nlines; j = j + 1)
```

```
            for (i= j - gap; i > 0; i = i - gap) {
                ig = i + gap
                if (compar(linptr(i), linptr(ig), linbuf) <= 0)
                    break
                call exchan(linptr(i), linptr(ig), linbuf)
                }
        return
        end
```

exchan csupán a két mutatót cseréli fel, paraméterként **linbuf**-ot is átveszi, hogy hívását általánosabb felhasználások esetén ne kelljen módosítani.

```
    # exchan — linbuf(lp1) és linbuf(lp2) felcserélése
        subroutine exchan(lp1, lp2, linbuf)
        character linbuf(ARB)
        integer k, lp1, lp2

        k = lp1
        lp1 = lp2
        lp2 = k
        return
        end
```

A **compar** negatív értéket ad vissza, ha az első paramétere kisebb, mint a második, nullát, ha a két paraméter egyenlő, és pozitív értéket, ha az első nagyobb, mint a második. Hogyan végezzük el az összehasonlításokat? A fő nehézség a használt jelkészlet örök problémája. Mindenesetre a FORTRAN-ban általában nincs arra biztosíték, hogy a **read** utasítással beolvasott „*a*" betű kisebb lesz a „*z*" betűnél. (Sok gépen valóban nem is lesz!) PL/1-ben **character** változókként tárolt betűk esetén a probléma nem merül fel — „*a*" kisebb, mint „*z*", de az ábécén belül is előfordulhatnak írásjelek. (Továbbá, a PL/1 **character** változói nem állnak elég bitből ahhoz, hogy lehetőséget adjanak speciális kódok, mint pl. **EOS** kifejezésére.) Az egész téma valóságos darázsfészek!

Az egyik lehetséges megoldás egy kétparaméteres rutin, a **lexord** bevezetése, amely minden karakterpárra megadja azok lexikális sorrendjét:

```
    lexord(c1, c2)
```

eredménye −1, ha **c1 < c2,** 0, ha **c1 = = c2** és +1, ha **c1 > c2.** Ily módon **compar** meghívhatja **lexord**-ot, ha meg akarja határozni két karakter sorrendjét. Természetesen **compar** egy hívásán belül **lexord**-ot általában csak egyszer kell meghívni. (Miért?) Így ez a megoldás nem annyira költséges, mint amilyennek első pillanatra látszik.

Egy másik megoldás, hogy **compar**-t valamilyen adott környezethez igazítjuk, mint ahogy majd mi is tesszük. Minden karaktert beolvasáskor azonnal a rendezési sorrendben neki megfelelő helyre képezünk le. Az összehasonlításokat ezután közvetlenül végezhetjük el. A végső kiírás előtt a karaktereket visszaállítjuk külső ábrázolásukra. Ez természetesen kompromisszum a hatékonyság és a portabilitás között. Ha a jelhalmaz ténylegesen össze van keverve, akkor ez a legolcsóbb szervezés. Jobb a karaktereket egyszer bevitelkor és egyszer kivitelkor transzformálni, mint minden

alkalommal, valahányszor összehasonlítjuk őket. A **compar** itt bemutatott változata akkor használható, ha a betűk és számok rendezési sorrendje növekvő.

```
# compar — linbuf (lp1) és linbuf (lp2) összehasonlítása
   integer function compar(lp1, lp2, linbuf)
   character linbuf(ARB)
  integer i, j, lp1, lp2

   i = lp1
   j = lp2
   while (linbuf(i) == linbuf(j)) {
        if (linbuf(i) == EOS) {
             compar = 0
             return
             }
        i = i + 1
        j = j + 1
        }
   if (linbuf(i) < linbuf(j))
        compar = — 1
   else
        compar = + 1
   return
   end
```

A **sort** szubrutinjainak szervezése olyan, hogy speciális alkalmazás esetén a tapasztalt felhasználó saját **compar** és **exchan** változatait használhatja benne, és emellett kihasználhatja a rendezési algoritmus által nyújtott előnyöket is. A program közvetlenül felhasználható egész számok rendezésére is: csupán arra van szükség, hogy a **compar** ne vegye figyelembe **linbuf**-ot.

Gyakorlatok

4.5 A rendezőprogram számottevő mennyiségű többletidőt használ fel a belső ciklusban. Végezzünk kísérleteket, amelyben az összehasonlítást és a cserét átvittük a **shell**-be. Mennyit javít ez a helyzeten? Dolgozzunk ki reális feltételezéseket a **sort** használatának gyakoriságára és a file-ok méretére vonatkozólag, és ezután döntsük el, szükség van-e a fenti változtatásra. □

4.6 Bővítsük ki a **sort**-ot olyan lehetőséggel, hogy a rendezési sorrendet meg lehessen változtatni:

```
sort    —r
```

nem növekvő, hanem csökkenő sorrendbe rendezzen. Hová kerüljön a sorrend-változtatást eredményező programrész: a **compar**-ba, a **shell**-be, a **ptext**-be vagy valahova máshova? □

4.4 GYORSRENDEZÉS

Az eredetileg C. A. R. HOARE által kifejlesztett *gyorsrendezés* az egyik legjobb rendezési módszer. Bár a legrosszabb esetben futási ideje n^2-tel is arányos lehet, elérhető, hogy ilyen eset ritkán forduljon elő, és az átlagos futási idő $n \cdot \log n$ legyen.

A gyorsrendezést legjobban rekurzív eljárásként lehet megadni. Az alapötlet az eredeti halmaz részekre bontása oly módon, hogy két csoportot képezünk: azon elemekét, amelyek egy, a halmazból választott tetszőleges elemnél kisebbek, és azokét, amelyek ennél nem kisebbek. Ezután ugyanezt a felosztást alkalmazzuk a két részhalmazra egymás után mindaddig, ameddig minden egyes részhalmaz csupán egyetlen elemből nem áll. Ha az összes részhalmazt felbontottuk, az eredeti halmaz rendezetté vált.

```
subroutine quick(v, i, j)

     if (i >= j)
        return
v(i) ...v(j) olyan felosztása, hogy
     v(i), v(i+1) ,..., v(k−1)≦v(k)≦v(k+1) ,..., v(j)
     legyen, ahol i≦k≦j
call quick(v, i, k−1)
call quick(v, k+1, j)
return
end
```

A **v** tömb rendezését egyszerűen a következő hívással intézhetjük el:

```
call quick (v, 1, n)
```

és ezzel készen is vagyunk. Mielőtt a részletekbe mennénk, nem ártana, ha az Olvasó néhány egyszerűbb esetet kézi úton kipróbálna, hogy meggyőződjön róla, valóban érti-e az alapvető vezérlési szerkezetet.

Az algoritmus lényege: „csoportosítsuk az elemeket úgy, hogy ...". A rekurzív algoritmus nem készít másolatot az eredeti **v** tömbről; minden tennivalót indexek mozgatásával intéz el, melyek megmutatják, hogy egy adott lépésben **v**-nek melyik intervallumát kell átrendezni. Ez azt jelenti, hogy a gyorsrendezéskor csak annyi többlettárolóra van szükség, amennyit a még fel nem osztott részhalmazokat leíró indexhatárokat tartalmazó verem igényel. Könnyen belátható, hogy ha minden egyes lépésben elsőként a „rövidebb" részhalmazzal foglalkozunk, a verem soha nem lesz ($\log_2 n$)-nél mélyebb. Még ha n eléri is az egymilliót, $\log_2 n$ csak 20, így a többlettárigény nem jelentős.

Gyakorlati okokból nem kívánunk rekurziót alkalmazni, nem mintha költséges lenne (hiszen a gyors rendezést az nem drágítja lényegesen, mivel a rekurzió nem a legbelső ciklusban van), hanem azért, mert a FORTRAN a rekurziót nem engedi meg. Ennek szimulálásához egy veremre van szükségünk, amely az eredeti tömb még fel nem osztott részhalmazainak indexeit tartalmazza. A nemrekurzív gyorsrendezés vázlata a következő:

```
subroutine quick(v, m, n)

lv(1) = m  # alsó határ
uv(1) = n  # felső határ
p = 1      # a szimulált rekurzió veremmutatója
while (p > 0)
    if (lv(p) >= uv(p))     # a részhalmaz egyetlen elemből áll
        p = p - 1           # a verem ürítése

    else {
            v(lv(p)),..., v(uv(p)) csoportosítása úgy, hogy valamilyen i-re
            v(lv(p)),..., v(i-1)≦v(i)≦v(i+1),..., v(uv(p)) legyen
            if (alsó csop. hosszabb) {  # a rövidebb kerüljön felülre
             az alsó vermelése p-nél
             a felső vermelése (p+1)-nél
            }
        else {
            a felső vermelése p-nél
            az alsó vermelése (p+1)-nél
            }
        p = p + 1      # veremmutató növelése
        }
return
end
```

A legfontosabb lépés a csoportosítás. Ebből a célból kiválasztunk egy tetszőleges „főelemet", **pivlin**-t, amely például a halmaz utolsó eleme lehet, és átrendezzük a sorokat a főelemhez viszonyítva. Az átrendezés a **v lv(p)** és **uv(p)** közti részhalmazán belüli elemeket érinti.

```
i = lv(p) - 1
j = uv(p)
pivlin = az utolsó sor, azaz line(j)
while (i < j) {
    i nő, amíg line (i) >= pivlin
    j csökken, amíg j <= i | line(j) <= pivlin
    # itt i és j egyenlő, vagy találtunk
    # egy rossz sorrendben levő párt
    if (i < j)      # sorrendcsere a páron belül
        line(i) és line(j) felcserélése
    }
# i és j egyenlő volt
# a pivlin főelem "középre" vitele
line(i) és az utolsó sor felcserélése
```

A tényleges program megírása megkülönböztetett gondosságot igényel. Beláttuk, hogy csak úgy tudunk eredményt elérni, ha állítások egy sorozatát — olyan kijelentéseket, melyeknek igaznak kell lenniük a program egyes helyein — előre leírjuk és a sorozatot azzal az állítással fejezzük be, hogy egy felosztásnak sikeresen vége van. Nem minden program igényli formális állítások írását és verifikálását, de a módszer alkalmas annak megértésére, hogy a program valójában hogyan működik. Ez a meg-

értés pedig nélkülözhetetlen ahhoz, hogy igazán bízhassunk a program hibátlanságában.

A felosztást úgy kezdjük, hogy főelemként a **linbuf(linptr(j))** utolsó elemet választjuk. Amikor a csoportosítás befejeződött, azt akarjuk, hogy

— *i* az **lv(p), uv(p)** zárt intervallumból vegyen fel valamilyen értéket;
— az *i*-nél kisebb indexű sorok egyike se legyen nagyobb, mint **linbuf(linptr(i))**;
— az *i*-nél nagyobb indexű sorok egyike sem kisebb, mint **linbuf(linptr(i))**.

Ezen kívül természetesen biztosnak kell lennünk abban, hogy nem lépjük túl a tömb határait és nem semmisítünk meg akaratlanul elemeket. A csoportosítás programja a következő:

```
i = lv(p) - 1
j = uv(p)
pivlin = linptr(j)          # a fősor
while (i < j) {
    for (i = i + 1; compar(linptr(i), pivlin, linbuf) < 0; i = i + 1)
            ;
        # linptr(lv(p))...linptr(i-1) a fősornál ≦ sorokra mutat
        # i≦uv (p), mert a fősor megállította
    for (j = j - 1; j > i; j = j - 1)
            if (compar(linptr(j), pivlin, linbuf) <= 0)
                break
        # linptr(j+1)...linptr(uv(p)-1) a fősornál≧sorokra mutat
        # j≧i≧lv(p), így nem futottunk túl a végén
    if (i < j)
            call exchan(linptr(i), linptr(j), linbuf)
        # i < j; i és j fordított sorrendű sorokra mutat
        # ha i < j, linbuf(linptr(i))≦fősor & linbuf(linptr(j)) ≧ fősor
    # i és j egyenlők voltak
    # a fősor átvitele az i. pozícióba
call exchan(linptr(i), linptr(uv(p)), linbuf)
    # helyes volt az i. sor fősorként való kiválasztása
```

A fenti állítások formálisan nem teljesek, de elegendőek ahhoz, hogy a program helyességében biztosak legyünk.

Most már összerakhatjuk a részeket, hogy megkapjuk a gyorsrendezés végleges változatát, a **quick** szubrutint:

```
# quick — szövegsorok gyorsrendezése
    subroutine quick(linptr, nlines, linbuf)
    character linbuf(ARB)
    integer compar
    integer i, j, linptr(ARB), lv(LOGPTR), nlines, p, pivlin, uv(LOGPTR)

    lv(1) = 1
    uv(1) = nlines
    p = 1
    while (p > 0)
        if (lv(p >= uv(p))          # csak egy elem van a részhalmazban
            p = p - 1               # a verem ürítése
```

```
        else {
            i = lv(p) - 1
            j = uv(p)
            pivlin = linptr(j)    # a fősor
            while (i < j) {
                for (i = i + 1; compar(linptr(i), pivlin, linbuf) < 0;
                    i = i + 1)
                    ;
                for (j = j - 1; j > i; j = j - 1)
                    if (compar(linptr(j), pivlin, linbuf) <= 0)
                        break
                if (i < j)              # pár rossz sorrendben
                    call exchan(linptr(i), linptr(j), linbuf)
                }
            j = uv(p)                   # fősor átvitele az i. pozícióba
            call exchan(linptr(i), linptr(j), linbuf)
            if (i-lv(p) < uv(p)-i) {    # vermelés úgy, hogy a
                                        # rövidebbet intézzük el előbb
                lv(p+1) = lv(p)
                uv(p+1) = i - 1
                lv(p) = i + 1
                }
            else {
                lv(p+1) = i + 1
                uv(p+1) = uv(p)
                uv(p) = i - 1
                }
            p = p + 1                   # a veremmutató növelése
            }
    return
    end
```

LOGPTR a legkisebb egész szám, amely legalább akkora, mint $\log_2$ **MAXPTR**, a rendezendő sorok maximális száma. Mint már említettük, ez csupán 20, még egymillió sorból álló file-ok esetén is.

Mint a legtöbb rendezési algoritmusnak, a gyorsrendezésnek is sok változata van. Itt csak néhányat említünk meg, a fejezet végén levő bibliográfia még többre utal.

A **quick** itt közölt formájában nem használja ki azt a tényt, hogy több elem is egyenlő lehet a főelemmel. Ha egy felosztás során az összes ilyen elemet összegyűjtjük, további felosztásokban már nem kell részt venniük. A „nagy főelem" algoritmus a csoportosításból a k_1 és k_2 értékeket adja vissza, melyekre

v(lv(p)) ,..., v(k1−1) < v(k1) =...= v(k2) < v(k2+1) ,..., v(uv(p))

és a következő felosztásokat az

lv(p) ,..., k1−1 és k2+1 ,..., uv(p)

részhalmazokból képezzük. Ez a módszer szavak listájához hasonló adatok rendezésénél gyorsabb, ha azok gyakran tartalmaznak ismételten előforduló elemeket.

Gyakorlat

4.7 Végezzünk kísérleteket a „nagy főelem" algoritmussal. Mennyivel gyorsabb olyan file-oknál, melyek jelentős mennyiségű ismétlődést tartalmaznak? Mennyivel lassúbb az eredeti gyorsrendezésnél olyan file-ok esetén, melyekben nincs vagy csak igen kevés ismétlődés van? □

4.8 Ha egy tömb már bármelyik sorrendben is rendezett, az utolsó elemnek főelemként való kiválasztása igen hátrányos: az algoritmust n^2 idejű eljárássá változtatja. (Miért?) Az egyik megoldás az, hogy a főelemet a halmaz közepén választjuk ki. Bonyolultabb, de hatékonyabb módszer, ha főelemként három vagy több elem mediánját választjuk. Vizsgáljuk meg részletesen ezeket a változatokat! □

4.9 A mi gyorsrendezésünk a ténylegesen szükségesnél több összehasonlítást végezhet. Keressünk olyan változatot, amely csökkenti a tényleges sorok összehasonlításának számát, még azon az áron is, hogy több adminisztratív műveletet végez, és nézzük meg, hogy milyen eltérések adódnak! □

4.5 NAGY FILE-OK RENDEZÉSE

A „nagy" jelző itt azt jelenti, hogy több adattal dolgozunk, mint amennyi a tárban egyszerre elfér; itt kezdenek a dolgok bonyolódni. Az ilyenfajta rendezést gyakran *külső rendezésnek* hívják, mivel az adatok egy részét ideiglenes közbenső file-okban kell tartani. Ennek ellentételeként, az előző pontban szereplő rendezést *belső rendezésnek* nevezik.

Nemcsak a belső, de a külső rendezés esetén is meglepően nagy számú változat áll rendelkezésre. Az alapötlet a legtöbb esetben egyszerű: a bemenet egyes részeit (amilyen nagyokat csak lehet) belső rendezéssel feldolgozzuk és közbenső file-okba másoljuk; minden ilyen részt egy-egy menetnek nevezhetünk. Ha már a teljes bemenet rendezett menetekre van osztva, a meneteket összefuttatjuk általában további közbenső file-okra. Végül az összes adatot egyetlen file-ba futtatjuk össze, és ez az utolsó menet a rendezett kimenet.

Nem minden operációs rendszer engedi meg tetszőleges számú file létrehozását, amit elvben ez a megoldás igényelt, ezért a szervezést bonyolíthatja a közbenső file-ok korlátozott száma. Azonban, ha dolgozhatunk is igen sok közbenső file-lal, a nagyszámú forrás összefuttatása megfelelő szervezést kíván, különben túlságosan lassúvá válik. (Tekintsük a szélső esetet: ha minden file az eredeti inputnak csak egyetlen sorát tartalmazza, mennyi időt vesz igénybe az összefuttatás a sorok számának függvényében?)

Az egyik legáttekinthetőbb rendezési eljárás az, amikor minden menetet különkülön file-ban helyezünk el, mindaddig, amíg a bemenő adatok el nem fogynak. Ezután az első m számú file-t egy új file-ban futtatjuk össze és az m számú file-t eltávolítjuk (m-et az összefuttatás *rendszámának* nevezzük, értéke rendszerint 3 és 7 között van). Ezt a folyamatot ismételjük a következő m számú file-lal, mindaddig, amíg csak egyetlen file-unk marad, és ez a rendezett kimenet. Ez az eljárás soha nem kezel m-nél több összefuttatandó file-t és esetenként egy outputfile-t.

A **sort** főrutinja ezt a stratégiát valósítja meg. Legnagyobb része a file-ok létrehozását, megnyitását, lezárását és eltávolítását végzi a megfelelő időpontokban.

```
# sort — szövegsorok külső rendezése
    character linbuf(MAXTEXT), name(NAMESIZE)
    integer gtext, makfil, min, open
    integer infil(MERGEORDER), linptr(MAXPTR), nlines
    integer high, lim, low, outfil, t

    high = 0
    repeat {              # a menetek kezdeti kialakítása
        t = gtext(linptr, nlines, linbuf, STDIN)
        call quick(linptr, nlines, linbuf)
        high = high + 1
        outfil = makfil(high)
        call ptext(linptr, nlines, linbuf, outfil)
        call close(outfil)
        } until (t == EOF)

    for (low = 1; low < high; low = low + MERGEORDER) {
        # összefuttatás
        lim = min(low+MERGEORDER-1, high)
        call gopen(infil, low, lim)
        high = high + 1
        outfil = makfil(high)
        call merge(infil, lim-low+1, outfil)
        call close(outfil)
        call gremov(infil, low, lim)
        }

    call gname(high, name)      # a végső rendcsinálás
    outfil = open(name, READ)
    call fcopy(outfil, STDOUT)
    call close(outfil)
    call remove(name)
    stop
    end
```

A **sort** összefuttatási fázisa két indexet használ: a **low**-t és a **high**-t, melynek a még aktív file-ok tartományát jelölik ki. A **high** értéke 1-gyel növekszik, majd a **low**-nál kezdődő **MERGEORDER** számú file a **high** által mutatott file-ban fut össze összefuttatásra, és ezt követően **low MERGEORDER**-rel növekszik. Amikor **low** utoléri **high**-t, az összefuttatás befejeződött, és az utolsó file-ban levő egyetlen menet a végső kimenetre másolódik.

A fejezet előző részében már megismertük **gtext**-et, **ptext**-et és **quick**-et, valamint a 3. fejezetben láttuk **fcopy**-t, amely egy file-t másol egy másikba. A **min** két szám minimumát határozza meg; FORTRAN-ban **min0**-lal lehet helyettesíteni.

A főrutinban a file-okra létrehozásuk sorrendjének megfelelő szám segítségével hivatkozunk. Új, ideiglenes file létrehozását adott szám esetén **makfil** végzi, amely **gname**-t használja a számnak egyedi, módszeresen létrehozott névvé való konvertálására.

```
# makfil — az n-hez tartozó új file létrehozása
    integer function makfil(n)
    character name(NAMESIZE)
    integer create
    integer n

    call gname(n, name)
    makfil = create(name, READWRITE)
    if (makfil == ERR)
        call cant(name)
    return
    end
```

A **gname** egy állandó névkezdetet (**stemp**) helyez el **name**-ben, majd jelsorozatként hozzáilleszti *n*-et. Ily módon a **sort** által használt ideiglenes file-ok nevei: **stemp1**, **stemp2** stb.

```
# gname — n azonosítójú egyedi file-név előállítása
    subroutine gname(n, name)
    character name(NAMESIZE)
    integer itoc, length
    integer i, junk, n
    string stemp "stemp"

    call scopy(stemp, 1, name, 1)
    i = length(stemp) + 1
    junk = itoc(n, name(i), NAMESIZE-i)
    return
    end
```

Emlékeztetünk rá, hogy **junk** az a „szemétláda", amelybe a fölösleges függvényértékeket írjuk be, amint ezt már a 2. fejezetből ismerjük.

A **gopen** és **gremov** sorrendben számozott file-ok halmazát nyitja meg és távolítja el. Figyeljük meg, hogy mindketten újra előállítják a file nevét, ahelyett, hogy bonyolult névátviteleket végeznének.

```
# gopen — low-tól lim-ig terjedő file-csoport megnyitása
    subroutine gopen(infil, low, lim)
    character name(NAMESIZE)
    integer i, infil(MERGEORDER), lim, low
    integer open

    for (i = 1; i <= lim-low+1; i = i + 1) {
        call gname(low+i-1, name)
        infil(i) = open(name, READ)
        if (infil(i) == ERR)
            call cant(name)
        }
    return
    end
```

```
# gremov — low-tól lim-ig terjedő file-csoport megszüntetése
  subroutine gremov(infil, low, lim)
  character name(NAMESIZE)
  integer i, infil(MERGEORDER), lim, low

  for (i = 1; i <= lim-low+1; i = i + 1) {
       call close(infil(i))
       call gname(low+i-1, name)
       call remove(name)
       }
  return
  end
```

Bármelyik időpontban legfeljebb **MERGEORDER** számú input, és egy outputfile van megnyitva. A munka folyamán valószínűleg ennél több ideiglenes file keletkezik, ezek azonban nincsenek egyidejűleg megnyitva. A **sort** feltételezi, hogy dinamikusan lehet file-okat létrehozni és méretüket a bemenet olvasása folyamán az igényeknek megfelelően beállítani, bár egyes rendszerekben ezen file-ok mérethatárait esetleg számszerűen meg kell adni. Lényeges, hogy a file-ok létrehozása és megszüntetése a felhasználó tudta nélkül történjen. Kevés dolog veszi el a felhasználó kedvét olyan gyorsan, mint az, hogy misztikus paraméterek felhasználásával munkafile-ok halmazait kell előállítania egy különben egyszerű feladat elvégzése érdekében.

Már csak a **merge** maradt magyarázat nélkül. A feladata elvileg egyszerű. Mivel az inputfile-ok rendezettek, minden file-ban az első sor a legkisebb. A **merge** ezek közül kiválasztja a legkisebbet, amely szükségképpen a legkisebb sor az egész csoportban is, és átmásolja a kimenetre. A kivitt sor helyére ugyanezen file következő sora lép és újra meghatározásra kerül a legkisebb sor. Amikor egy file-on **EOF**-hez értünk, a megfelelő menet befejeződött. Amikor minden menetnek vége, a **merge** befejezte munkáját.

A fő kérdés az, hogy hogyan válasszuk ki hatékonyan a mindenkori legkisebb sort? A kézenfekvő módszer, az éppen vizsgált **MERGEORDER** számú sorban való lineáris keresés akkor fogadható el, ha **MERGEORDER** kicsi, de egy jobb algoritmussal és adatszerkezettel célszerűbben is eljárhatunk.

Az egyik legjobb módszer a soroknak „halmok"-ba való csoportosítása. A halomnak megvan az az előnyös tulajdonsága, hogy legkisebb eleme közvetlenül megtalálható, és egy új elemet a halom megfelelő pozíciójában annyi idő alatt tudunk elhelyezni, ami a méretével csupán logaritmikusan növekszik. Képletesen egy halmot bináris faként képzelhetünk el (amelyben minden elemnek legföljebb két utódja van), és amelyben minden elem kisebb vagy egyenlő utódjaival. Programozási szempontból a halmot a legegyszerűbb egy **h** tömbként ábrázolni oly módon, hogy a k-adik elem leszármazottait a $2k$-adik és a $(2k+1)$-edik pozícióban tároljuk. Ily módon a **h(1)** nem nagyobb **h(2)**-nél és **h(3)**-nál; **h(2)** nem nagyobb **h(4)**-nél és **h(5)**-nél, és általában **h(k)** nem nagyobb **h(2k)**-nál és **h(2k+1)**-nél. A halom legkisebb eleme **h(1)**.

A halmok segítségével az összefuttatási folyamat a következő:

```
egy sor olvasása minden file-ből
a halom kialakítása
while (van még input) {
        a legkisebb sor kivitele [halom (1)]
        egy új sor behozatala halom (1)-be
        halomátrendezés: az új sor átvitele
        a halomban neki megfelelő helyre
        }
```

Az első pozícióban a legkisebb sor áll. Ez az elem kiíródik, majd helyére egy újabbat olvasunk be, amely ezután a halom megfelelő helyén tárolódik (a halom átrendezése). A kezdő halom létrehozható oly módon, hogy **quick** segítségével rendezzük a sorokat, mivel egy rendezett tömb mindig halom. (Miért?) Ennek az ellenkezője általában nem igaz. (Miért?) Igaz, hogy a rendezés a szükségesnél egy kicsivel több munkát jelent, de a különbség alig észrevehető. Miért írnánk erre külön programot?

```
# merge — infil(1)...infil(nfiles) összefuttatása outfil-ben
   subroutine merge(infil, nfiles, outfil)
   character linbuf(MERGETEXT)
   integer getlin
   integer i, inf, lbp, lp1, nf, nfiles, outfil
   integer infil(MERGEORDER), linptr(MERGEORDER)

   lbp = 1
   nf = 0
   for (i = 1; i <= nfiles; i = i + 1)  # egy sor olvasása minden file-ból
      if (getlin(linbuf(lbp), infil(i)) ¬= EOF) {
         nf = nf + 1
         linptr(nf) = lbp
         lbp = lbp + MAXLINE       # hely a leghosszabb sornak
         }
   call quick(linptr, nf, linbuf)    # kezdő halom elkészítése
   while (nf > 0) {
      lp1 = linptr(1)
      call putlin(linbuf(lp1), outfil)
      inf= lp1 / MAXLINE + 1       # file-index számítása
      if (getlin(linbuf(lp1), infil(inf)) == EOF) {
         linptr(1) = linptr(nf)
         nf = nf - 1
         }
      call reheap(linptr, nf, linbuf)
      }
   return
   end
```

A bonyolult tárkezelés elkerülése érdekében a **merge** minden esetre a lehetséges leghosszabb sor számára biztosít helyet. Ez egyszerűsíti annak eldöntését, hogy egy adott sor melyik file-hoz kapcsolódik, mivel csupán a sor kezdőpozícióját kell elosztani a maximális sormérettel.

A halomba való elhelyezés során a legfelső elemet összehasonlítjuk utódaival. Ha ez az elem egyik utódjánál sem nagyobb, akkor a megfelelő helyen van és készen vagyunk. Ha valamelyiknél nagyobb, akkor az elemet kicseréljük utódai közül a kisebbel, és az eljárást a fa következő szintjén megismételjük, azaz az elemet új pozíciójában hasonlítjuk össze utódaival. Végül az elem a fában saját helyére vándorol.

Könnyű kimutatni, hogy a halomba való beillesztés ideje **MERGEORDER** logaritmusával arányos, míg a lineáris keresés természetesen **MERGEORDER**-rel egyenesen arányos. A halomkezelő eljárás programja csupán néhány sorral hosszabb a lineáris változaténál, és a **MERGEORDER** tipikus értékei esetén sokkal gyorsabb lehet.

```
# reheap — linbuf((linptr(1)) beillesztése a megfelelő helyre
    subroutine reheap(linptr, nf, linbuf)
    character linbuf(MAXTEXT)
    integer compar
    integer i, j, nf, linptr(nf)

    for (i = 1; 2 * i <= nf; i = j) {
        j = 2 * i
        if (j < nf)            # a kisebb utód keresése
            if (compar(linptr(j), linptr(j+1), linbuf) > 0)
                j = j + 1
        if (compar(linptr(i), linptr(j), linbuf) <= 0)
            break              # megvan a kívánt pozíció
        call exchan(linptr(i), linptr(j), linbuf)      # vándoroltatás
        }
    return
    end
```

Figyeljük meg, hogy a **quick**, a **merge** és a **reheap** egyaránt helyesen dolgozik, ha adatok nélkül kerül hívásra. Ez kizárólag azért van, mert a program összes ciklusában a ciklusváltozó vizsgálata a ciklus elején van elhelyezve, és nem a végén. Egy program általános megbízhatósága szempontjából jó előjel, ha határesetei külön erőfeszítés nélkül megbízhatóak.

Célszerű, ha az olvasó újra átnézi a **sort** szerkezetét, és megfigyeli annak funkcionális modulokra való felosztását. A rendezés, az összefuttatás, az összehasonlítás, a bemenet és a kimenet mindegyike külön kicserélhető, anélkül, hogy a program többi részét zavarná.

Gyakorlatok

4.10 Egyidejűleg hány közbenső file-területre van szükség az *n* karaktert tartalmazó input rendezéséhez? □

4.6 TOVÁBBFEJLESZTÉS

Ha az alapváltozat már működik, gyorsíthatjuk a **sort**-ot és több feladat ellátására is alkalmassá tehetjük. Mivel ez a könyv eszközökről, és nem algoritmusokról szól, és bennünket inkább a működésbeli tökéletesítések érdekelnek, mégis először tekintsünk át néhány, a hatékonyságot illető megfontolást.

A **sort**-tal kapcsolatban két különböző input esetén végeztünk méréseket. Az első 6900 karaktert tartalmazó 270 sorból állott, a másodikban 1330 sor és 34 800 karakter volt. A **linbuf** számára 10 000 karaktert engedtünk meg, így az első adathalmazt egyetlen közbenső file-lal, összefuttatási fázis nélkül lehetett feldolgozni. A második csoport először négy menetet igényelt, melyek aztán egy ötödik file-ban futottak össze a végső kimásolás előtt. A központi egység ideje a két csoporthoz a következő:

teljes idő	**6,5 s**	**36,9 s**
összes B/K	**94,1%**	**91,6%**
compar	**4,6**	**6,1**
quick	**0,9**	**1,2**
merge	**...**	**0,3**
reheap	**...**	**0,3**

Az összes többi 0,15%-nál kevesebbet tesz ki.

A B/K idők különösen szembetűnőek. Amint ezt már említettük és újra hangsúlyozni fogjuk, a FORTRAN karakter B/K általánosságát drágán kell megfizetni. A fenti időkön biztosan javíthatunk specializált rutinok használatával. Azonban még tetszőleges tökéletesítés esetén is minden bizonnyal a B/K fog dominálni. Mivel az adatok ki- és bemásolása nem kerülhető el, olyan algoritmust kell találnunk, amely a közbülső B/K-t csökkenti.

Gyakorlatok

4.11 Amikor már csak legfeljebb **MERGEORDER** számú közbülső file van hátra, egy teljes menet megtakarítható, ha az összefuttatást egy közbülső file helyett közvetlenül az eredmény-file-ba irányítjuk. Ennek speciális esete, ha az eredeti bemenet teljes egészében elfér a tárban; ekkor nincs szükség semmilyen összefuttatásra és közbülső file-ra. Módosítsuk **sort**-ot úgy, hogy ezeket az eseteket hatékonyan kezelje. A fenti mérések alapján mennyivel gyorsítják fel ezek a változtatások **sort**-ot? □

A **sort** futási idejét erősen befolyásolja a bemenő adatokon való áthaladások száma, amely viszont a kezdeti bejáráskor létrehozott menetek hosszától függ. Több tárral mindig nyerhetünk hosszabb meneteket, de a több tárterület nem áll mindig rendelkezésünkre.

4.12 A rendelkezésre álló tárterület jobb kihasználásának egyik lehetősége például az, hogy egynél több karaktert helyezünk el egy egész változóban. Módosítsuk **sort**-ot úgy, hogy több karaktert tömörítsen bevitelkor egy egész változóban, és kivitelkor állítsa vissza az eredeti helyzetet. Mennyit és melyik rutinokban kell változtatnunk? Adott tárméret esetén milyen sebességnövekedés érhető el? □

Helyettesítéses kiválasztás

Hosszú kezdeti menetek létrehozásának különösen elegáns módja a „helyettesítéses kiválasztás". A tárban egyszerre tárolt adatokat ugyanúgy rendezzük, mint eddig. Ezután azonban, amint **ptext** kivisz egy sort, helyére rögtön beolvasunk egy újabbat. Ha az új sor nagyobb vagy egyenlő, mint az a sor, amelynek helyére kerül, akkor része lehet a pillanatnyilag kifelé tartó menetnek! Ha kisebb, akkor ebben a menetben nem vihető ki; meg kell tartani a következő menetig.

Ez a szervezés jelentős előnyökkel jár. Véletlenszerű tartalommal rendelkező file-ok esetében kiderül, hogy a menetek várható hossza a tárméret kétszerese, így megtakaríthatjuk az adatok egy teljes bejárását. Természetesen az eredmény még jobb, ha a file-ok részlegesen már rendezettek.

A helyettesítéses kiválasztás gondos tárgazdálkodást igényel, ha a sorok nem egyforma hosszúak. Ami még fontosabb, szükség van egy olyan algoritmusra is, amely gyorsan megtalálja a helyettesítő sor megfelelő pozícióját a tárban levő, már rendezett elemek között. A megfelelő hely megtalálása érdekében túlságosan lassú lenne végignézni az összes aktuális sort. A megoldás valamilyen halomszerű adatstruktúra alkalmazása, ahol a keresett pozíciót nem lineáris, hanem logaritmikus időigénnyel találhatjuk meg. Ennek részleteivel itt nem foglalkozunk, a témát meglehetős alapossággal tárgyalja D. E. KNUTH: „The Art of Computer Programming" című könyv Sorting and Searching című harmadik kötetében.

Mit tehetünk, ha az általunk használt rendszer nem engedi meg túlságosan sok file létrehozását? A probléma egyik lehetséges megkerülése az úgynevezett „kiegyensúlyozott kétutas összefuttatás". Tegyük fel, hogy négy közbenső file-t használhatunk. Az első fázisban a meneteket fölváltva az egyes és a kettes file-ban tároljuk, ameddig a bemenő adatok el nem fogynak. Ekkor az egyes és a kettes file-on levő meneteket összefuttatjuk oly módon, hogy a kiírásra kerülő menetek (feleannyi és mindegyik mégegyszer akkora) felváltva a hármas és a négyes file-ra íródnak. Ezután a hármas és a négyes file-on levő menetek kerülnek összefuttatásra és visszaírásra az egyes és a kettes file-ba, és a menetek most már az eredetinek a négyszeresei. A folyamat addig folytatódik, ameddig minden adat nem áll összefuttatásra egyetlen, teljesen rendezett file-ban, melyet azután kimásolunk. Mivel a menetek hossza minden átfutáskor duplázódik, a menetek száma pedig feleződik; az eljáráshoz $[\log_2] r$ lépésre van szükség, ahol r az első fázisban létrehozott menetek száma.

A kiegyensúlyozott kétutas összefuttatást tetszőleges számú (3 vagy több) közbenső file-ra is általánosíthatjuk. A rendelkezésre álló file-okat két, nagyjából egyenlő csoportra osztjuk és az egyesítés a két csoport között oda-vissza történik.

Gyakorlatok

4.13 Valósítsuk meg a kiegyensúlyozott kétutas összefuttatást: A fő problémát az jelenti, hogy meg kell jegyeznünk minden egyes menet végét minden egyes file-on. Hasonlítsuk össze a kiegyensúlyozott kétutas összefuttatás programjának bonyolultságát és futási idejét a **sort**-ével! □

Foglalkozzunk most a rendezőprogram funkcionális tökéletesítésével. Említettük már, hogy különösen hasznos a **sort**-ot szűrő formájában előállítani. Valószínű azonban, hogy a hagyományosabb módon is hasznosnak fog bizonyulni.

4.14 Alapfeltételezésként a **sort** a standard inputról olvas. Módosítsuk oly módon, hogy ha híváskor file-neveket is megadunk, akkor bemenetét az adott file-okból vegye:

sort *file1 file2 ...*

rendezze a megnevezett file-okban levő adatokat. Igen sok program esetében bevált tervezési gyakorlat az, hogy az input a megadott file-okról vagy ilyenek hiányában a standard input perifériáról származik. Soha se hagyjuk figyelmen kívül ezt a lehetőséget! □

4.15 Egészítsük ki a rendezést azzal a lehetőséggel, hogy az outputfile-t név szerint adhassuk meg a standard output helyett. Tegyük lehetővé, hogy az inputfile-ok egyike output is lehessen. Célszerű-e a végső összefuttatást közvetlenül ide irányítani? □

4.16 Már rendezett file-ok összefuttatásán valósítsuk meg a **–m** funkciót:

sort –m *file1 file2* ...

összefuttatja (a rendezettnek feltételezett) adatokat a file-okból a standard outputra. A

sort –m

parancs file-nevek nélkül értelmetlen. Ennek ellenére elképzelhető-e ennek a parancsnak valamilyen logikus értelmezése? □

4.17 Vezessük be a **sort**-hoz a **–r** funkciót, amely megfordítja a rendezés sorrendjét. Ha megoldottuk az ezzel kapcsolatos 4.6 gyakorlatot, szükségesnek látjuk-e a kód elhelyezésével kapcsolatos akkori döntésünket megváltoztatni? □

4.18 Vezessük be a **–d** funkciót, melynek hatására a **sort** szótári sorrendbe rendez: a kis- és nagybetűk azonosnak minősülnek, így pl. „a" és „A" együtt jelenik meg, és nem választja el őket a kisbetűk teljes sorozata. Elegendő-e a következő sorrend definiálása: **a**<**A**<**b**<**B**...? Célszerű-e, ha a szótári sorrend a rendezés alapfeltételezés szerinti működése? Mit kell tenni a speciális jelekkel — pont, vessző stb. — kapcsolatban? Mi a helyzet a számjegyekkel? □

4.19 Vezessük be a **–n** funkciót: a sor elején álló numerikus jelsorozat egy esetleges előjellel együtt aritmetikai értéke szerint rendeződik, míg a sor többi részének rendezése a szokásos. A tiszta numerikus sorok az ilyenfajta adatok speciális esetei. Működik-e a rutinunk a számok méretétől függetlenül? □

4.20 Vezessünk be olyan funkciókat, amelyek a soroknak a soron belüli mezők szerinti rendezését teszik lehetővé. Szükségünk lesz minden egyes mező kezdetének és végének megjelölésére, és valószínűleg arra is, hogy a mezők rendezési szabályát (numerikus, szótári sorrendű, fordított sorrendű stb.) egymástól függetlenül adhassuk meg. Az igazi problémát itt nem a működőképes program elkészítése jelenti, hanem a funkciók olyan módon való megtervezése, hogy megadásuk a felhasználó számára kényelmes legyen. Gondoljunk arra, hogy egy mezőnek karakterpozíciók segítségével való megadása nem könnyű, különösen, ha a bemenő jelsorozat nem jól elkülönülő oszlopokban érkezik. □

*

Ezzel már a kereskedelmi rendezőprogram-csomagok bonyolultságának közelébe értünk, de legalább szerény lépésekkel jutottunk el idáig, és minden lépésnél egy-egy újabb hasznos eszközt állítottunk elő. Alapfeltételezésként azonban a **sort** még mindig a standard inputból vett szövegsorokat rendez, és a standard outputra küldi az eredményeit. Ez mindig könnyen használható eszköz azok számára, akik csupán szöveges információt akarnak rendezni. A leggyakoribb és legegyszerűbb műveletek könnyen megjegyezhetők, ill. specifikálhatók kell hogy legyenek, és nem szabad, hogy állandóan szükségünk legyen egy kézikönyvre.

*

4.21 Egyes rendszerekhez nagy teljesítményű, de bonyolult rendezőprogram-generátor tartozik, amely hatékony rendezőprogramot hoz létre nagyméretű, „éles" feladatokhoz. Ha az Olvasó rendszerében létezik ilyen, tervezzen és valósítson meg egy olyan nyelvet, amely az alkalmi felhasználók számára megkönnyíti a szükséges rendezőprogram előállítását! □

4.7 A FUNKCIÓK SZÉTVÁLASZTÁSA: UNIQUE

A rendezés egyik általános célja az egyes elemek összes előfordulásainak egy helyre gyűjtése, hogy azután csoportonként legyenek kezelhetők. Néha egy csoportból csupán egyetlen elemet akarunk megtartani, például amikor egy szövegben előforduló összes szó listáját állítjuk elő. Minden bizonnyal könnyű a **sort**-ot olyan funkcióval bővíteni, amelyik kihagyja egy input sor ismételt előfordulásait. (Hova kellene ezt a programrészt beilleszteni?) Megkérdőjelezhető azonban, hogy ezt a feladatot a **sort**-on belül kell-e elvégezni?

Ez a kérdés a jó eszközök tervezése szempontjából alapvető területet érint — a tevékenységek megfelelő szétválasztását. Mely funkciókat kell egyetlen programba sűríteni, és melyek alkossanak külön programot? Az imént említett tevékenység hatékonysági meggondolásokból is a rendezőprogramba kívánkozik, mivel ezáltal az adatok egyszeri átnézése megtakarítható. Ami azonban ennél is fontosabb, két sor „azonosságának" eldöntése az összehasonlító függvénytől függ, amelyet természetszerűleg a rendezéshez előírt funkcióparaméterek határoznak meg. Ha a rendezést és az ismétlődések kihagyását egyetlen programban egyesítjük, biztosítva van, hogy az összehasonlítás konzisztens.

Miért lenne szükség két külön programra, amikor a feladatot egyetlen, alig valamivel bonyolultabb program is megoldja? Az egyik nyomós ok az lehet, hogy valaki igényelheti az egyik tevékenységet a másik nélkül is. Ha szétválasztjuk az ismétlődések elhagyását a rendezéstől, olyan feladatokat is el tudunk végezni, melyek megoldása a két funkció egyesítése esetén nem lehetséges. Valóban, szükségünk lehet annak ismeretére, hogy mely sorok *nem* fordulnak elő ismételten, és *mely sorok ismétlődnek;* vagy esetleg *meg akarjuk számlálni* az egymással szomszédos ismétlődéseket. Ha a rendezés, az ismétlődések elhagyása és a számlálás egyetlen programba kerül, akkor a rendezőprogram bonyolultabbá válik és emiatt elképzelhető, hogy nem akarjuk az adatokat az ismétlődések kiszűrése előtt rendezni.

Az egyes tevékenységek idő előtti összevonása hiba. Legalábbis kezdeti állapotában, egy program célszerűen csak egyetlen tevékenységet végezzen. A későbbiekben bizonyára több funkció megvalósítására is sor kerül, de ezeknek szorosan kell egymáshoz kapcsolódniuk. Ily módon, amikor a felhasználók a programok egymással való kombinálásának új lehetőségeit vetik fel, nem fordulhat elő, hogy néhány hasznos műveletet lehetetlenné tettünk a felhasználók szándékairól alkotott téves elképzelésünk miatt. Ebben saját tapasztalataink is iránymutatóak. Éveken át a rendezést és az ismétlődések kiszűrését különálló programok valósították meg. Végül a hatékonyság központi kérdéssé vált, és a két tevékenységet összevontuk: a **sort**-ot kibővítettük egy funkcióval, amely a szomszédos ismétlődések kihagyását tette lehetővé (bár természetesen az ismétlődést kihagyó régi funkció megmaradt, és továbbra is gyakran használják). Ugyanakkor kezdetben senki nem érzékelte, hogy ez a kombináció célszerű. A tanulság: tartsuk az egyes tevékenységeket külön addig, ameddig nem tudjuk, hogy hogyan célszerű azokat kombinálni.

A következőkben bemutatjuk a **unique**-ot, amely elhagyja a szomszédos ismétlődéseket. Leggyakrabban a **sort**-tal együtt használják, de önmagában is elég fontos ahhoz, hogy különálló program legyen.

```
# unique — a szomszédos ismétlődések elhagyása
    character buf1(MAXLINE), buf2(MAXLINE)
    integer equal, getlin
    integer t

    t = getlin(buf1, STDIN)
    while (t ¬= EOF) {
        call putlin(buf1, STDOUT)
        for (t= getlin(buf2, STDIN); t ¬= EOF; t = getlin(buf2,
            STDIN))
            if (equal(buf1, buf2) == NO)
                break
        if (t == EOF)
            break
        call putlin(buf2, STDOUT)
        for (t = getlin(buf1, STDIN); t ¬= EOF; t = getlin(buf1,
            STDIN))
            if (equal(buf1, buf2) == NO)
                break
        }
    stop
    end
```

Mely határeseteket kell kipróbálni, hogy a **unique** helyes működésében megbízhassunk?

Gyakorlatok

4.22 A **unique** általunk adott változata a program vezérlési szerkezetét használja fel azon puffer megjelölésére, amelyben azt a sort tároljuk, amellyel a többi sort folyamatosan összehasonlítjuk. Írjuk át a programot úgy, hogy egyetlen belső ciklust és egyetlen kapcsolót tartalmazzon, amely felcseréli a két puffer szerepét. Írjuk át a programot úgy is, hogy a sort egyszerűen átmásoljuk **buf2**-ből **buf1**-be, valahányszor az összehasonlítás nem egyezést mutat. A három változat közül melyiket tartjuk legjobbnak? □

4.23 Bővítsük ki a **unique**-ot a **−n** funkcióval, amely az eredeti bemenő sorokhoz prefixként hozzáilleszti az előfordulások számát. Az

unique −n

parancs

a
a
b

input esetén a következő outputokat állítsa elő:

2 a
1 b

Miért kell a számlálónak *megelőznie* a sort? □

4.24 Kombináljuk a **translit**-ot, a **sort**-ot és a **unique**-ot (a **−n** funkcióval) olyan csatlakozó programsorozattá, amely egy szöveghez megadja az egyes

szavak előfordulási gyakoriságait, gyakoriság szerint csökkenő sorrendbe rendezve. □

4.25 Kombináljuk **translit**-ot, **sort**-ot (a **−d** funkcióval) és **unique**-ot olyan csatlakozó programsorozattá, amely átvizsgál egy programot az egyes szavak együttes kis- és nagybetűs előfordulását keresve, mint pl. **SIZE** és **size.** □

4.26 Írjuk meg a **common** nevű programot, amely összehasonlítja két rendezett szöveges file sorait. A

common *file1 file2*

parancs az eredményt három oszlopban írja ki: a csak *file1*-ben előforduló sorok, a csak *file2*-ben előforduló sorok és a mindkettőben előforduló sorok. Megengedjük a **−1, −2** és **−3** nem kötelező paraméterek használatát is, amely csakis a megfelelő oszlop nyomtatását engedélyezi, így pl. a

common −3 *file1 file2*

parancs csak a két file közös sorait nyomtatja. Hasonló módon, a

common −1 *file1 file2*

parancs azon sorokat nyomtatja, amelyeket az első file tartalmaz, de a második nem. Ha csak egy file paramétert adunk meg, *file2*-ként a standard inputot kell venni. □

*

Mire jó a **common?** Tegyük fel, hogy rendelkezésünkre áll egy angol értelmező szótár. Tekintsük a következő programcsatlakozásokat:

```
concat file1 file2 file3... |
 translit A—Z a—z |
  translit ¬a—z @n |
   sort |
    unique |
     common −2 szótár
```

Ez egyesíti a file-ok egy halmazát **(concat)**, a szavakat kisbetűsekké konvertálja **(translit)**, elhagyja az írásjeleket és a szóközöket, minden szót külön sorban helyez el **(translit)**, rendezi őket **(sort)**, elhagyja az ismétlődéseket **(unique)** és kiválasztja azokat, amelyek előfordulnak az eredeti file-okban, de a szótárban nem **(common)**.

Mire utal az, ha egy szó előfordul a szövegben, de hiányzik a szótárból? Igen — az ilyen szavak helyesírási hibák szempontjából rendkívül gyanúsak. Ez a csatlakozó programsorozat az első nyers változata lehet egy helyesírási hibákat kiszűrő programnak. Egyáltalán nem végez tökéletes munkát, viszont pillanatok alatt összerakható meglevő alkatrészekből, és kitűnő alapot szolgáltat egy bonyolultabb eljáráshoz. Ezért hozunk létre eszközöket.

*

4.27 Hogyan tökéletesítené a helyesírásihiba-kereső programot? Milyen kísérleteket kellene elvégezni, mielőtt hozzáfognánk a „tökéletesítéshez?" □

4.28 A helyesírásihiba-kereső eredménye gyakran főként a szakmai zsargon szavaiból áll, olyanokból, mint pl. **byte** vagy **translit.** Ha a valóban hibás szavakat kiszűrtük az output-ból, ami megmarad, az egy szöveg speciális szavainak a szószedete lehet. Hogyan módosítanánk a programcsatlakozási sorozatot úgy, hogy az a szószedetbe tartozó szavakat kiküszöbölje a helyesírási hibák későbbi feldolgozásából? □

4.8 PERMUTÁLT INDEX

Ha már van egy rugalmas rendezőprogramunk, ezt más programok felhasználhatják. Ebben a részben egy ilyen alkalmazást tárgyalunk, a permutált index (gyakran a „keyword-in-context"* rövidítéseként KWIC indexnek is nevezik) létrehozására alkalmas programot. A permutált index minden szót (vagy hasonlóan hasznos *szövegegységet*) eredeti környezetében sorol fel, a szavak szerint rendezve, és olyan formában, hogy a kulcsszavak azonos pozíción kezdődnek. Például ez a mondat a következő eredményt adja:

Például ez	*a mondat a következő eredményt adja:*
ez a mondat	*a következő eredményt adja: Például*
következő eredményt	*adja: Például ez a mondat a*
mondat a következő	*eredményt adja: Például ez a*
adja: Például	*ez a mondat a következő eredményt*
mondat a	*következő eredményt adja: Például ez a*
Például ez a	*mondat a következő eredményt adja:*
eredményt adja:	*Például ez a mondat a következő*

A program egyik lehetséges szerkezeti felépítése a következő:

minden input sorra
 a sor minden szavára {
 permutáljuk ciklikusan a sort úgy, hogy a szó a sor elejére kerüljön
 vigyük ki egy közbenső file-ba
 }
rendezzük a közbenső file-t
a közbenső file minden sorára {
 az első szót „permutáljuk vissza" a sor közepére
 nyomtassuk ki a sort
 }

A folyamat három független program csatlakozásából álló sorozatként fogható fel:

ciklikus permutálás | **sort** | *„visszapermutálás" és nyomtatás*

de ennek egy adott rendszerben való megvalósítása nagymértékben függ attól, hogy milyen eszközök állnak rendelkezésre. A programcsatlakozás előnye, hogy a **sort** már jól kidolgozott formában megvan, a másik két rész pedig egyszerű.

A **kwic** megírásának másik lehetséges módja az, hogy **sort**-ot önálló programként hívjuk be abból a programból, amely a ciklikus permutálást és a visszapermutálást közönséges szubrutinok hívásával végzi el. Ez utóbbi módszer feltételezi, hogy az operációs rendszer megengedi egy programnak egy futó programból való elindítását, és azt, hogy ennek lefutása után a vezérlés az indító programba kerüljön vissza. Ennek szerkezete a következő módon érzékeltethető

ciklikus permutálások a **temp1** *file-ba*
futtatás **(„sort <temp1 >temp2")**
visszapermutálás **temp2**-*ből*

* Magyarul: kulcsszó a környezetben *(A lektor).*

A **sort** fekete doboz marad, a teljes folyamat mégis egyetlen programon belül játszódik le.

Bárhogyan is szervezzük a későbbiekben a programot, szem előtt kell tartanunk, hogy az *eredeti szerkezet* ne változzon. A megoldás egyes részeit, a lehető legnagyobb mértékben függetlennek akarjuk tartani, ezért kezdettől fogva úgy járunk el, mintha a programot a lehető legszigorúbb feltételeket szabó rendezésben kívánnánk megvalósítani (mint amilyen pl. a programcsatlakozás). Ily módon a legkisebb az esélye annak, hogy figyelmünket kijátszva az egyes modulok között „mellékutas" információcsere jöjjön létre. A tényleges modulszerkezettel kapcsolatos döntéseket addig késleltessük, ameddig csak lehet, hogy ezáltal választási lehetőségeinkben a legnagyobb szabadságot tartsuk fenn magunknak.

A részletes szerkezeti kialakítást az Olvasóra bízva bemutatjuk a ciklikus permutálás és a „visszapermutálás" rutinjait, melyeket programcsatlakozásban való felhasználás céljára írtunk. A ciklikus permutálás rutinja a következő:

```
# kwic — a kulcsszó előállítása
  character buf(MAXLINE)
  integer getlin

  while (getlin(buf, STDIN) ¬= EOF)
      call putrot(buf, STDOUT)
  stop
  end
```

A munkát a **putrot** végzi; ez minden sorban felkutatja a kulcsszavakat. Egy kulcsszó betűkből vagy számokból álló jelsorozat, melyből az írásjelek, mint pl. zárójelek, vesszők stb. kimaradnak. Ezeket úgy kell kihagyni, hogy a velük szomszédos szavak kinyomtatáskor azonos pozíción, a megfelelő oszlop elején kezdődjenek. A **putrot** megkeresi minden szó elejét, azaz az első alfanumerikus karaktert, és meghívja **rotate**-t ezen sor oly módon való kiírására, hogy ez a karakter a sor elejére kerüljön.

```
# putrot — kulcsszóval kezdődő sorok létrehozása
  subroutine putrot(buf, outfil)
  character type
  character buf(ARB), t
  integer i, outfil

  for (i = 1; buf(i) ¬= NEWLINE; i = i + 1) {
      t = type(buf(i))
      if (t == LETTER | t == DIGIT) {     # alfanumerikus szó
          call rotate(buf, i, outfil)      # kezdődik i-nél
          t = type(buf(i+1))
          for ( ; t == LETTER | t == DIGIT; t = type(buf(i+1)))
              i = i + 1
          }
      }
  return
  end
```

```
# rotate — a permutált sor kivitele
    subroutine rotate(buf, n, outfil)
    character buf(ARB)
    integer i, n, outfil

    for (i = n; buf(i) ¬= NEWLINE; i = i + 1)
        call putch(buf(i), outfil)
    call putch(FOLD, outfil)
    for (i = 1; i < n; i = i + 1)
        call putch(buf(i), outfil)
    call putch(NEWLINE, outfil)
    return
    end
```

A **rotate** megjelöli az eredeti sor végét (a sor szétvágásának helyét) és egy **FOLD** karaktert iktat be, amely egy normál szövegben nagy valószínűséggel nem fordul elő. A **FOLD**-ot a visszapermutáló program használja a permutált sorok visszaállítására. Ily módon, ha a **FOLD** karaktert a $ (dollárjel) karakterrel szimbolizáljuk, akkor a

most itt az ideje

input sor a következő négy output sort eredményezi:

most itt az ideje$
itt az ideje$most
az ideje$most itt
ideje$most itt az

A **type** függvény eldönti, hogy egy adott karakter betű, számjegy vagy más valami-e, eredményül **LETTER**-t, **DIGIT**-et, vagy magát a karaktert adja.

```
# type — karaktertípus meghatározása
    character function type(c)
    character c
    integer index
    string digits "0123456789"
    string lowalf "abcdefghijklmnopqrstuvwxyz"
    string upalf "ABCDEFGHIJKLMNOPQRSTUVWXYZ"

    if (index(lowalf, c) > 0)
        type = LETTER
    else if (index(upalf, c) > 0)
        type = LETTER
    else if (index(digits, c) > 0)
        type = DIGIT
    else
        type = c
    return
    end
```

Emlékeztetünk rá, hogy az **index** függvény meghatározza egy karakter pozícióját egy karaktersorozaton belül, ill. zérust ad vissza, ha a karaktert nem találta meg.

Bár a **type** függvény hatékonyabb lehet, ha kihasználjuk egy adott jelkészlet numerikus tulajdonságait, a jelen változat előnye, hogy a jelkészlettől független, és ezért különböző gépekre változtatás nélkül átvihető.

A csatlakozó programok sorának másik végén az **unrot** áll, amely visszapermutálja és kinyomtatja a permutált sorokat, egyenes margót képezve a **MIDDLE** oszlopban. Először a sor első felét helyezi el a **MIDDLE+1** pozíciótól kezdve, majd ha szükséges, a sor végén ugyanazon sor elejére tér át. Ezután a sor második fele következik, **MIDDLE−1**-től visszafelé haladva, végül pedig törlődnek a sorvégi szóközök.

```
# unrot — kwic által rotált sorok visszapermutálása
   character inbuf(MAXLINE), outbuf(MAXOUT)
   integer getlin, index
   integer i, j

   while (getlin(inbuf, STDIN) ¬= EOF) {
      for (i = 1; i < MAXOUT; i = i + 1)      # üres sor
         outbuf(i) = BLANK
      j = MIDDLE
      for (i = 1; inbuf(i) ¬= FOLD & inbuf(i) ¬= NEWLINE;
         i = i + 1) {
         j = j + 1              # másolás a FOLD-ig
         if (j >= MAXOUT − 1)
            j = 1
         outbuf(j) = inbuf(i)
         }
      if (inbuf(i) == FOLD) {     # a sor másik felének másolása
         j = MIDDLE               # visszafelé haladva
         for (i = index(inbuf, NEWLINE) − 1; i > 0; i = i − 1) {
            if (inbuf(i) == FOLD)
               break
            j = j − 1
            if (j <= 0)
               j = MAXOUT − 2
            outbuf(j) = inbuf(i)
            }
         }
      for (i = MAXOUT − 2; i > 0; i = i − 1)
         if (outbuf(i) ¬= BLANK)    # sorvégi szóközök elhagyása
            break
      outbuf(i+1) = NEWLINE        # a sor megfelelő lezárása
      outbuf(i+2) = EOS
      call putlin(outbuf, STDOUT)
      }
   stop
   end
```

Vegyük észre, hogy **unrot** akkor is ésszerűen viselkedik, ha a bemenetben nincs **FOLD** karakter, mivel a **NEWLINE**-t és a **FOLD**-ot egyaránt figyeli. Ez a defenzív programozás egy újabb példája: úgy írjuk meg a programot, hogy kisebb balesetekkel

meg tudjon birkózni. A balesetek természetesen különfélék lehetnek, és nem szabad megengedni, hogy a védekezés eluralkodjon, de ebben az esetben a biztonság nem költséges.

Gyakorlatok

4.29 Miért írtuk a következőt:

```
for (i = index(inbuf, NEWLINE) − 1; i > 0; i = 1 − 1) {
    if (inbuf(i) == FOLD)
        break
    ...
```

ahelyett, hogy:

```
for (i = index(inbuf, NEWLINE) − 1; i > 0 & inbuf(i) ¬= FOLD;
    i = i − 1) {
    ...
```

(*Útmutatás:* l. a **compress**-t a 2. fejezetben.) □

4.30 A **kwic** és az **unrot** nem megfelelően kezeli a tabulátort vagy visszalépést tartalmazó szöveget. Ugyancsak figyelmen kívül hagyják azt a lehetőséget, hogy a szöveg **FOLD** karaktereket is tartalmazhat. Oldjuk meg ezeket a problémákat! □

4.31 Módosítsuk **kwic**-et úgy, hogy ne vágjon el szót a kimeneten! □

4.32 Hamar észrevesszük majd, hogy olyan szavakat, mint **a, az, és** stb. feleslegeses a **kwic** indexben szerepeltetni. Továbbá célszerű lehet olyan listákat előállítani, amelyek csupán bizonyos szavakat tartalmaznak. Teremtsük meg a lehetőségét annak, hogy ezeket meg lehessen adni akár egy „kihagyandó", akár egy „kiírandó" file (vagy mindkettő) formájában, ahol az első a **kwic** indexből elhagyandó, a második pedig az indexben kizárólagosan szereplő szavakat tartalmazza. (A 8. fejezet tartalmaz néhány alapvető táblázatkezelő eljárást.) □

4.33 Módosítsuk **kwic**-et úgy, hogy a standard input helyett több file kezelésére is alkalmas legyen, akárcsak a **print**. Egy nagyobb anyag esetén, amely több inputfile-ból áll, hasznos lehet pontosan tudni, hogy egy adott sor az inputban hol fordul elő. Hozzuk létre a **−t** funkciót, amely a kimenő sorok eredetét azonosítja, pl. a file nevével és egy sorszámmal. Célszerű-e ezt alapfeltételezésként megvalósítani? □

A **kwic** program segítségével gyorsan (és nem elegáns módon) ellenőriztük, hogy programunk összes változóját deklaráltuk-e. (A deklarálatlan változók rendszerint elírásból származnak.) Ez a megoldás azonban kézi munkát is igényel és ezért nem olyan jó, mint amilyennek lennie kellene.

4.34 Ha a fordítóprogramunk ezt nem végzi el, készítsünk programot, amely ellenőrzi, hogy programjainkban minden változó deklarálva van-e. □

4.35 Írjunk programot egy szöveg kereszttreferencia-listájának elkészítésére (a szöveg gyakran, ha nem is mindig, egy program). Ez azt jelenti, hogy minden szövegegységre listázzuk ki azon sorok sorszámát, amelyekben erre az egységre való hivatkozás található. □

BIBLIOGRÁFIAI MEGJEGYZÉSEK

A belső és külső rendezéssel kapcsolatos alapvető irodalom D. E. KNUTH: *„The Art of Computer Programming"* c. könyvének 3. kötete: Sorting and Searching (Addison-Wesley, 1973). Ez igen sokféle rendezési módszer pontos leírását és részletes elemzését tartalmazza. A belső rendezéssel kapcsolatos másik forrásmunka R. P. RICH: *„Internal Sorting Methods Illustrated with PL/1 Programs"* c. könyve (Prentice-Hall, 1972), amely PL/1 programokat, valamint ezek részletes, futási időre, illetőleg tárigényre vonatkozó elemzését tartalmazza.

A mi gyorsrendezésünket a *Knuth* által a „Structured Programming with goto statements" c. cikkben (*Computing Surveys,* 1974. dec.) bemutatottból vettük át; a módszert KNUTH R. SEDGEWICK-nek tulajdonítja. A gyorsrendezés más változatait tárgyalja R. LOESER a „Some performance test of quicksort and descendants" c. cikkében (*CACM,* 1974. márc.). Ebben a cikkben minden eljáráshoz FORTRAN-programok tartoznak. Tanulságos lehet a FORTRAN-t a RATFOR-ral összevetni.

D. L. PARNAS az „On the criteria to be used in decomposing systems into modules" c. cikkében (*CACM,* 1972. dec.) egy permutált index program modularizálását tárgyalja. Hasonlítsuk össze programjának szerkezetét a mienkével!

A helyesírási hibák kiszűrésére használható csatlakozó programok sorozata S. C. JOHNSON ötlete volt. A helyesírási hibák szótár alkalmazása nélkül való felderítésével foglalkozik R. MORRIS és L. L. CHERRY a „Computer detection of typographical errors" c. cikkben (*IEEE Transactions on Professional Communication,* 1975. márc.).

5. fejezet

Szövegminták

Emlékszünk még a Bevezetésben szereplő **format** keresőre? Abban a feladatban olyan programra volt szükség, amely a **format** szót — bárhol forduljon is elő egy input sorban — megtalálja és az ilyen sorokat kinyomtatja.

Természetesen senki sem akar olyan programot írni, amely egy speciális FORTRAN utasítás megtalálására alkalmas, még olyat sem, amely FORTRAN programokban való keresésre korlátozott. A **format** szó speciális *szövegminta,* olyan programra van szükség, amelynek paraméterként megadható, hogy milyen szövegmintát keressen. Legyen **find** ez a program. Így a

find *minta*

parancs hatására kinyomtatódnak mindazok az input sorok, melyekben az adott minta előfordul. Ezután például a **format** utasításokat a

find format

parancs segítségével találhatjuk meg.

Erre egyesek úgy érvelhetnek, hogy tulajdonképpen egy szövegszerkesztőre van szükségünk, amely szövegminták keresésére is alkalmas. Egy ilyen szerkesztőre valóban szükségünk van és ezt létre is fogjuk hozni a következő fejezetben. Emellett azonban igényünk van a **find**-ra is. Ennek oka az, hogy egyes célokra a szerkesztőprogram *túl általános.* Pillanatnyi feladatunk megoldása céljából meg kell hívnunk a szerkesztőt, egyenként közölni vele, hogy mely file-okat akarjuk feldolgozni, majd minden file-ra ismételten ki kell adnunk a kereső utasítást. Ez az előkészítés túl bonyolult. Ugyanakkor a **find,** a lehető legkevesebb felhajtással, pontosan azt végzi el, amire szükségünk van.

Alig múlik el munkanapunk úgy, hogy ne használnánk széleskörűen a **find**-ot. A legkézenfekvőbb alkalmazás olyan kérdések megválaszolása, mint pl. „Mikor fordult először elő a **break** utasítás?” vagy „Hol vannak hivatkozások egy adott változóra?”. Ugyancsak jól használható szűrőként egy terjedelmes output-ból való válogatásra, például a következő programcsatlakozás formájában:

program | find error

ami kiválasztja azokat az üzeneteket, amelyek az „error” szót tartalmazzák. (Ez egy szövegszerkesztő segítségével sokkal nehezebben lenne elvégezhető.)

A **find**-ot használtuk fogalmazási stílusunk javítására is. Feltűnhetett, hogy a könyvben gyakran fordul elő az „egyszerű” szó és ennek származékai. Ez önmagában érthető, mivel az egyszerűség programok esetén erénynek számít. Ugyanakkor a túl gyakori használat rontja a szó kifejezőerejét, ezért átfuttattuk a szöveget a

```
find egyszerű
```

parancs segítségével (ez kiszűri az „egyszerűen"-t és az „egyszerűség"-et is), majd bizonyos előfordulásokat megfelelő szinonimákkal helyettesítettünk. Egy alkalommal még meg is számláltuk őket a

```
find egyszerű | linecount
```

programcsatlakozás segítségével.

5.1 SZÖVEGMINTÁK

Nem lenne értelme a **find**-ot minden elképzelhető szövegminta felismerésére felkészíteni. Túlzásba eshetünk például olyan módon, hogy olyan program létrehozását erőltetjük, amely a FORTRAN **format** utasításának összes szabályos előfordulását megkeresi, megengedve a kulcsszóban előfordulható szóközöket, a többszörös folytatási sorokat stb. Azt is kiköthetnénk továbbá, hogy csakis a **format** utasítások kerüljenek kinyomtatásra és azok a sorok nem, melyek a **format** jelsorozatot karakterláncként vagy egy változó neveként tartalmazzák. Mire azonban megoldjuk az olyan ravasz értékadó utasítások kezelését, mint pl. a

```
10 format(x5h) = b(i)
```

elkészítettük a FORTRAN fordító szintaxis elemzőjének legnagyobb részét!

A mi **find** programunk nyilván nem foglalkozik ilyen patologikus esetekkel, de nézzük meg, milyen gyakran fordulnak ezek elő? Mikor írta az Olvasó utoljára a **format**-ot **for mat**-nak? Mikor használta utoljára azonosítóként? Ha az általunk kívánt sorok mindegyike kinyomtatásra kerül, nem nagy baj, ha néhány felesleges is megjelenik. Hacsak nem akarunk teljesen precízek lenni, megvonhatunk egy ésszerű határt. Egy-egy alkalmazás esetén elfogadunk néhány gyengeséget az alkalmazási terület kiszélesítése érdekében. Egy eszköz legtöbb alkalmazója hajlamos a kompromisszumokra, boldog lesz, ha a feladatot 90%-ban elvégezzük.

Egy olyan egyszerű jelölésrendszerre szorítkozunk, melyet több interaktív szövegszerkesztőnél és más mintaillesztő programnál alkalmaztak. A jelölésrendszer gazdaságos és emellett meglepően sokoldalú. Gyakorlásként felvetünk néhány hasznos általánosítást is.

Egy szövegminta lehet olyan egyszerű, mint az **a** betű, vagy egyszerű elemek bonyolultabb konstrukciója, mint pl. a **format** jelsorozat. Tetszőleges szövegminta felépítéséhez néhány szabály megjegyzése elegendő.

Bármely jel, mint pl. az **a** betű, egy szövegminta, amely a feldolgozás alatt álló szövegben a vele azonos karakterre illeszkedik. Jelek sorozata, mint pl. **123** vagy **format** olyan minta, amely ezen karaktersorozatnak az input egy során belül annak bármely előfordulására illeszkedik.

Egy mintáról akkor mondjuk, hogy „illeszkedik" a szövegsor egy részéhez, ha a sor tartalmazza a minta egy előfordulását. Például, az **aa** minta egyszer az első pozíciónál illeszkedik az **aabc** sorhoz, az **aabcaaba** sorhoz két helyen, az **aaaaaa** sorhoz pedig öt (egymást átlapoló) helyen illeszkedik. Az illesztés sorról sorra haladva történik, vagyis semmilyen minta illesztése nem léphet át sorhatárt. A szövegmintákat *konkatenálhatjuk:* két egymás után álló szövegminta egy olyan új mintát ad,

amely illeszkedik mindazon mintákra, melyek első felére az első, második felére a második komponens illeszkedik. Karakterek sorozata példa a konkatenált mintára.

Bár nem nehéz olyan programot írni, amelyik jelsorozatok előfordulását figyeli (és első lépésként hasznos is), ennek korlátai rövidesen kitűnnek. Ezért a **find**-ot további lehetőségekkel bővítjük: olyan minták keresésével,

— melyek karakterek osztályaihoz illeszkednek,
— melyek a soron belül csak meghatározott pozícióban elhelyezkedő mintákhoz illeszkednek, vagy
— melyek meghatározatlan hosszúságú szöveghez is illeszkednek.

Ezen általánosabb minták kifejezése érdekében néhány karaktert arra kell lefoglalnunk, hogy bizonyos típusú szövegmintákat reprezentáljanak vagy azokat elhatárolják. Például a **?** karaktert szövegmintaként arra használjuk, hogy az új sor kivételével bármely karakterhez illeszkedjék. Így pl. az **x?y** minta illeszkedik **x+y**-ra, **xay**-ra, **x?y**-ra és egyéb hasonló jelsorozatokra.

A **?**-t és a többi lefoglalt karaktert gyakran metakarakternek nevezik. Törekszünk arra, hogy metakarakternek olyan karaktereket válasszunk, melyek szövegekben általában ritkán fordulnak elő, mégis megtörténhet, hogy egy metakarakter eredeti értelmű előfordulását akarjuk vizsgálni. Ezért a metakarakterek speciális jelentését „kikapcsolhatjuk", ha eléjük az @ felhívójelet írjuk, mint azt a **translit** karakterkonverterben, a 2. fejezetben tettük. Így @**?** az eredeti **?**-hez illeszkedik, és @@ az eredeti @ karakterhez.

A **[** metakarakter azt jelzi, hogy a következő **]**-ig terjedő karakterek egy *karakterosztályt* képviselnek, azaz olyan szövegmintát, amely a listához tartozó minden egyes karakterhez illeszkedik. A karakterosztályok leírásakor ugyanazt a jelölést használjuk, mint a **translit from** karakter sorozatainál:

— **[aA]** **a**-ra vagy **A**-ra illeszkedik,
— **[a—z]** bármely kisbetűre,
— **[¬a]** „**a**" kivételével bármely karakterre

stb. Az egyetlen különbség, hogy kényelmi szempontból kikötjük: a „negált" karakterosztályok, mint amilyet az utolsó példa határoz meg, sohasem illeszkednek az újsor-karakterhez. A felhívójel használata osztályokon belül is megengedett, ha a karakterosztály a ¬, **—**, @, vagy **]** valamelyikét tartalmazza.

Két további metakarakter nem explicit karakterekhez, hanem soron belüli pozícióhoz illeszkedik. A **%** a sor kezdetéhez illeszkedik: **% abc** olyan minta, amely csak akkor illeszkedik **abc**-hez, ha az egy sor első három karaktereként fordul elő. Hasonlóképpen, **$** a sor végén álló újsor-karakterhez illeszkedik: **abc$** csak akkor illeszkedik **abc**-hez, ha az a sor utolsó három karaktere az új sor előtt. Ezek természetesen együtt is használhatók: **%abc$** olyan sorra illeszkedik, amely csupán az **abc** karakterekből áll, **%$** pedig csak az üres sorokhoz illeszkedik (vagyis olyan sorokhoz, melyek egyetlen újsor-karakterből állnak.)

Az egyetlen karakterre illeszkedő fenti szövegminták bármelyike (a **%** és **$**-ral megjelöltek kivételével) után egy * írható, amelyik az egykarakteres minta zérus vagy többszörös folyamatos előfordulásához illeszkedik. Az így kapott mintát az egykarakteres minta *lezártjá*nak nevezzük. Például **a*** zérus vagy akárhány **a**-hoz, **aa*** egy vagy több **a**-hoz, **[a—z]*** pedig zérus vagy akárhány kisbetűből álló tetszőleges sorozathoz illeszkedik.

Mivel egy minta lezártja a minta zérus vagy több példányához illeszkedik, el kell döntenünk, hogy több lehetőség esetén melyik illeszkedést választjuk. A **find**-nak tulajdonképpen elég annyit tudnia, hogy van-e legalább egy illeszkedés a sorban, de később használni akarjuk az illesztéssel kiválasztott részsorozatot. Az tűnt a legcél-

szerűbbnek, ha **find** a lehető leghosszabb sorozathoz illeszti a mintát, még akkor is, ha egy üres sorozat szintén kielégítené az illeszkedési szabályokat.

Ily módon **[a–zA–Z]*** egy teljes szóhoz (ami üres sorozat is lehet), **[a–zA–Z] [a–zA–Z]*** egy teljes szóhoz (egy vagy több betű, de nem üres sorozat), és **?*** egy teljes sorhoz (amely üres sorozat is lehet) illeszkedik. Bármely többértelműséget, amely azzal kapcsolatban merülhet fel, hogy egy adott minta egy sor melyik részéhez illeszkedik, oly módon oldunk fel, hogy a legszélső bal oldali karakterrel kezdődő, lehető leghosszabb illeszkedő sorozatot vesszük. Ily módon **[a–z] [a–z0–9]** a sor legszélső bal oldali FORTRAN azonosítójához, **(?*)** két zárójel közötti tetszőleges sorozathoz, és **??*** egy vagy több karakterből álló teljes sorhoz illeszkedik (de nem illeszkedik egy önmagában álló újsor-karakterből álló sorhoz).

Egyetlen minta sem lép át új sorba. Ez a legtöbb esetben természetes és hasznos, mivel például megakadályozza, hogy egy gondatlanul elhelyezett **?*** „megegye" az összes bemenő adatot.

Technikailag az általunk bevezetett szövegminták a jól kidolgozott elmélettel rendelkező *reguláris kifejezések* egy alosztályát alkotják. Az általános reguláris kifejezések jellemzője, hogy alternatívák megadását és minták zárójelezését teszik lehetővé, így pl. **x(a | bc)y** vagy **xay**-hoz, vagy **xbcy**-hoz illeszkedik. (A zárójelek és a függőleges vonal metakarakterek lesznek.) Ezen általánosabb minták nagyobb kifejezőképességének ára, hogy bonyolultabbak. A mi esetünkben a nagyobb kifejezőerő nem éri meg a vele járó bonyolultságot, de néhány, ezzel kapcsolatos kérdést a későbbiekben még tárgyalni fogunk.

Gyakorlatok

5.1 *a)* Írjunk olyan szövegmintát, amely azon szavakhoz illeszkedik, melyekben az é, e, á, a magánhangzók ebben a sorrendben fordulnak elő, mint pl. a „kényszerpálya" vagy „bélyegpárna" szavakban. *b)* Írjunk olyan szövegmintát, amely azon szavakra illeszkedik, melyek egy fordítva tartott zsebszámológépen megjeleníthetők. (Ezek rendszerint a BEhILOS betűk, a 8341705 számjegyeknek megfelelően.) □

5.2 Írjunk olyan szövegmintákat, melyek PL/1 azonosítókhoz, Cobol azonosítókhoz és az olvasó által használt assembly nyelv által megengedett azonosítókhoz illeszkednek. Hogyan használhatnánk fel a **translit, sort, unique** és **find** programokat egy program összes azonosítójának és kulcsszavának kiírására? Kiszűrhetők-e egyszerű módon a kulcsszavak? (Útmutatás: nézzük meg a 4. fejezetben található helyesírási hibákat ellenőrző programot.) Megéri-e az eredmény a befektetett munkát? □

5.3 A legtöbb nyelv rögzíti az azonosítók maximális hosszát, FORTRAN-ban ez például 6 karakter. A fent definiált szövegminták segítségével tudunk-e olyan mintát írni, amelyik legfeljebb hatkarakteres FORTRAN azonosítókhoz illeszkedik? □

5.4 Felhasználható-e a **find** az összes üres sor eltávolítására? A megjegyzéseket tartalmazó összes kártya eltávolítására egy FORTRAN programból? Ezek megszámlálására? Az összes megjegyzés eltávolítására egy PL/1 programból? Felhasználható-e arra, hogy egy **end** utasítás után eltávolítsa az összes sort? □

5.5 A fenti definíciók alapján értelmesek-e az alábbi minták? Ha nem, miért nem, ha igen, mi a jelentésük?

a**
aa*

```
a%b$c
%*a
[¬ @t]*[ @t]
%$
%
*
@@t
□
```

5.2 A MEGVALÓSÍTÁS

Most, hogy már ismerjük azokat a szövegmintákat, melyeket keresni akarunk, elkezdhetjük a program felépítését. A részletek ismerete nélkül is látható, hogy szükségünk lesz a **pat** tömbre, ahol a mintát tároljuk. A program felső szintjének szerkezete a következő:

```
# find — minta keresése szövegben
    character arg(MAXARG), lin(MAXLINE), pat(MAXPAT)
    integer getarg, getlin, getpat, match

    if (getarg(1, arg, MAXARG) == EOF)
        call error ("hívás: find minta")
    if (getpat(arg, pat) == ERR)
        call error("hibás minta")
    while (getlin(lin, STDIN) ¬= EOF)
        if (match(lin, pat) == YES)
            call putlin(lin, STDOUT)
    stop
    end
```

A **getpat** a paramétert felhasználva a keresendő mintát elhelyezi **pat**-ban. A **match** deríti fel, hogy a **lin** bemenő soron belül előfordul-e valahol illeszkedés és a **YES/NO** választ adja vissza.

Gyakran előfordul, hogy nemcsak megírni tudjuk a programot a végcél teljes ismerete nélkül, de ki is tudjuk azt próbálni. Ezt tettük a **find** esetében is. A **getpat** és a **match** „üres" változatait használva meggyőződhettünk róla, hogy a szövegsorok olvasása és írása megfelelően történik (vagyis hogy az **EOF** detektálása jókor megy végbe), és hogy a beolvasott sor **lin**-ben levő belső ábrázolását legalábbis **getlin** és **putlin** megfelelően kezeli. A **getpat** egy kisebb változtatással kipróbálta a hibaüzenetet is. A **match** egy végsőkig leegyszerűsített formája — amely csak egy szó eleji „**a**" betűt ismert fel — kimutatta, hogy a szelektív módon való sorkiírás is működik.

Ez igen egyszerűen hangzik, de ily módon meglepően sok hibát lehet igen korán felderíteni. Nagy szoftver vállalkozások esetén a hibák jelentős része abból ered, hogy a rendszer egyes részei nem a kívánt módon működnek együtt, annak ellenére, hogy a részletes kapcsolódási specifikációt kezdettől fogva minden érdekelt ismerte. Igen sok hiba átcsúszik az egyes rutinokban végzett részletes próbákon, és csak akkor kerül felszínre, ha a rutin a program többi részével kezd együttműködni.

Ezért természetesnek tűnik, hogy a legmagasabb, legnagyobb mértékben integrált szinten kell a próbákat kezdeni, mivel a legtöbb hiba amúgy is ezen a szinten

bukik ki. A tesztelést olyan korán *kell elkezdeni, amilyen korán csak lehetséges,* még mielőtt a ténylegesen működő program jó része meg lenne írva. Ezt a megközelítést felülről lefelé haladó *tesztelésnek* nevezik, ami a felülről lefelé haladó *tervezés* és *programozás* természetes általánosítása. Az ilyenkor használt üres programokat imitált rutinoknak nevezik. A **find**-ot ezzel a módszerrel építettük fel, és próbáltuk ki, fokozatosan haladva egyik résztől a másikig és ez a megközelítés bevált. Több ostoba hiba ellenére (melyek közül néhányat bemutattunk) a program megírása és belövése rövid ideig tartott.

Mivel illeszkedés egy soron belül akárhol előfordulhat, a legegyszerűbbnek tűnik az illesztést két részre bontani. A **match** egy soron belül tetszőleges helyen előforduló illeszkedést keres oly módon, hogy ismételten hívja az **amatch**-et, amelyik egy rögzített helyen, az **i**-edik pozíción való ún. kötött illeszkedést vizsgálja. Ez különválasztja az illeszkedés tényének ellenőrzését annak eldöntésétől, hogy legközelebb milyen illeszkedést próbáljunk találni.

```
# match — illeszkedés keresése bárhol a soron belül
  integer function match(lin, pat)
  character lin(MAXLINE), pat(MAXPAT)
  integer amatch
  integer i

  for (i = 1; lin(i) ¬= EOS; i = i + 1)
      if (amatch(lin, i, pat) > 0) {
          match = YES
          return
          }
  match = NO
  return
  end
```

Az **amatch** valamilyen módon megmutatja, hogy az illesztett jelsorozat hol helyezkedik el, vagy zérust ad vissza, ha nem volt illeszkedés. Bár a későbbi programoknak tudniok kell, hogy a minta a szöveg melyik részéhez illeszkedett, a **find** csupán az illeszkedés tényével törődik.

Ha egyelőre figyelmen kívül hagyjuk a metakaraktereket, az **amatch**-nek a mintát a bemenettel karakterről karakterre kell összehasonlítania mindaddig, amíg eltérést nem talál (amikor is zérust ad vissza), vagy sikeresen el nem éri a minta végét (amikor is az input soron következő pozícióját adja vissza, amely biztosan nem a nulladik). Ennek alapján az **amatch** legelső változata a következő:

```
# amatch metakarakterek nélkül
  integer function amatch(lin, from, pat)
  character lin(MAXLINE), pat(MAXPAT)
  integer from, i, j

  i = from
  for (j = 1; pat(j) ¬= EOS; j = j + 1) {
      if (lin(i) ¬= pat(j)) {
```

```
            amatch = 0
            return          # nincs illeszkedés
            }
        i = i + 1
        }
    amatch = i
    return                  # sikeres illeszkedés
    end
```

A minta karakterei a **pat** tömb egymásutáni elemeiben helyezkednek el.

A **?**, **%** és **$** metakarakterek nem jelentenek nagy problémát. A karakterosztályok azonban felvetik az ábrázolás kérdését. Nyilvánvaló, hogy nem akarjuk az **[a–z]**-hez hasonló rövidítéseket a bemenet minden soránál az egyes karakterpozíciókra külön-külön értelmezni. Úgy tűnik, hogy a szövegminták bonyolultsága valamilyen kódolás bevezetését indokolja. Ennek megfelelően egyszer végigmegyünk a mintán, gondosan ellenőrizzük, hogy nem tartalmaz-e szabálytalan specifikációt, kifejtjük a rövidítéseket, és a mintát valamilyen kényelmesebb formára hozzuk. Mivel a **find**-dal meglehetősen nagy file-ok feldolgozására számítunk, szeretnénk, ha az illeszkedést vagy nem illeszkedést minél gyorsabban el tudnánk dönteni. A minta kódolása egy általános elv specifikus példája — minél több időt szánunk az adatok előfeldolgozására, felhasználásuk annál gyorsabbá tehető.

Mintáinkat oly módon ábrázoljuk, hogy a **pat**-ban minden mintatípust egy speciális kód reprezentálja. A tényleges karaktereket két elem ábrázolja: A **CHAR** indikátor a **j**-edik pozíción, és maga a karakter a **(j+1)**-edik. A **%**, **$** és **?** metakaraktereket rendre a **BOL, EOL** és **ANY** elemek fejezik ki. A rövidítések kifejtése után a karakterosztályokat vagy a **CCL** (**[...]** esetén) vagy az **NCCL** (**[¬...]** esetén) jelöli, melyet az osztályhoz tartozó karakterek száma és maguk a karakterek követnek. A minták lezártjait nem tárgyaljuk addig, ameddig a könnyebb feladatokat meg nem oldottuk. Eszerint a

%[¬x]?[0–9]x$

minta kódolt formája a következő:

BOL NCCL 1x ANY CCL 10 0 1 2 3 4 5 6 7 8 9 CHAR x EOL

(A kódban előforduló első 1-es és a 10-es: szám és nem karakter.) A minta kódolt formájának előállítását a **getpat** és az általa hívott rutinok végzik el, melyekre rövidesen visszatérünk.

Mivel ilyen bonyolulttá tettük a minták ábrázolását, érdemes egy külön rutint írni egyetlen karakter összehasonlítására, hogy az **amatch** kezelhető méretű maradjon. Az **omatch** megvizsgálja, hogy egy bemenő jel egyezik-e minta aktuális pozíciójában álló karakterrel és ha igen, a bemenet pozícióját a megfelelő mértékben előre állítja. Így **amatch** arra szorítkozhat, hogy végighalad a mintán, megfelelően szinkronizálva azt az illesztés alatt álló szöveggel. Most már megírhatjuk az **amatch** újabb változatát:

```
# amatch néhány metakarakterrel
    integer function amatch(lin, from, pat)
    charakter lin (MAXLINE), pat (MAXPAT)
    integer omatch, patsiz
    integer from, i, j

    i = from
    for (j = 1; pat(j) ¬= EOS; j = j + patsiz(pat, j))
        if (omatch(lin, i, pat, j) == NO) {
            amatch = 0
            return              # nincs illeszkedés
            }
    amatch = i
    return                      # sikeres illeszkedés
    end
```

Az **omatch** mindent elintéz a minták lezártján kívül. A **patsiz** meghatározza a **pat** egy elemének hosszát, így lehetővé teszi a következő elem elérését. Mindkét rutinhoz visszatérünk majd, amikor a minták lezártjaival foglalkozunk.

A legnagyobb problémát a lezártak okozzák. A jelen fejezet végén levő szövegmódosító programban és a 6. fejezetben levő szerkesztőprogramban olyan részek vannak, melyek az illesztett szöveget valami mással helyettesítik. Ennek érdekében a legcélszerűbb, ha a lehetségesek közül mindig a lehető leghosszabb mintát illesztjük. Így egy * előfordulása esetén a mintát ciklikusan ismételjük mindaddig, amíg az illeszkedés meg nem szűnik. A feldolgozást ezúttal ettől a ponttól folytatjuk a minta következő elemének és az input sor folytatásának lehetőség szerinti illesztésével.

Mi történik akkor, ha a minta maradéka nem illeszkedik a sor hátralevő részéhez? Ez *nem* feltétlenül jelenti azt, hogy nem volt illeszkedés. Például a **b*b** minta illeszkedik ugyan a **bb** sorhoz, de csak akkor, ha **b*** csupán az első **b**-t „fogyasztja" el (vagy az ez előtt álló üres sorozatot). Ez azt jelenti, hogy valahányszor egy illesztés nem sikerül, vissza kell mennünk az utolsó lezártig, azt eggyel rövidíteni kell, majd a minta maradékának illesztését újra meg kell kísérelni. Csak akkor adhatjuk fel, ha ez a kísérlet úgy mond csődöt, hogy a lezártat már üres sorozathoz illesztettük.

És még ekkor sem biztos, hogy készen vagyunk, hiszen egy mintában több lezárt is lehet. (Ne felejtsük el, azt mondtuk, hogy az *utolsó* lezártig megyünk vissza.) Így, a minták korrekt kezelése érdekében rendszeresen visszalépésekre kényszerülünk, melyekkel bejárjuk a lezártak összes lehetséges illesztését, és végül vagy találunk illeszkedést, vagy tényleg nincs ilyen.

A visszalépések megoldásának egyik módja a rekurzív eljárás. Ennek előnye, hogy a fordítóprogram előállítja az adminisztrációhoz szükséges könyvelési tevékenységek nagy részét, melyek egy nem rekurzív programot elbonyolítanak. Például **amatch** megírható a következő formában:

```
# amatch — lezártak kezelését végző rekurzív változat (pszeudokód)
    integer function amatch(lin, from, pat)
    offset = from                    # következő feldolgozatlan karakter
    for (j = 1; pat(j) ¬= EOS; j = j + patsiz(pat, j))
        if (pat(j) == CLOSURE) {  # egy lezártra utaló bejegyzés
            j = [az ismétlődő minta indexe]
            for (i = offset; lin(i) ¬= EOS; i = i + 1)   # illesszünk
                if (omatch(lin, i, pat, j) ==NO)          # amennyit
                    break                                 # csak lehet

                # i az első nem illeszthető karakterre mutat
                # vessük össze a minta folytatását a sor maradékával;
                # minden eltérésnél a lezárt csökken eggyel
            for (j=[ismétlődő minta]; i >= offset; i = i - 1) {
                k = amatch(lin, i, pat(j))
                if (k > 0)   # sikeres illesztés a maradékban
                    break
                }
            offset = k        # ha k == 0 sikertelen,
            break             # ha k > 0 sikeres illesztés
            }
        else if (omatch(lin, offset, pat, j) == NO) {  # nem lezárt
            amatch = 0
            return            # sikertelen illesztés
            }                 # nem lezárt minta esetén
        # különben omatch sikeres volt
    amatch = offset
    return
```

A rekurziót a

```
k = amatch (lin, i pat(j))
```

utasítás jelenti.

Ezt a változatot PL/1-ben úgy valósíthatjuk meg, hogy az **amatch** eljáráshoz a **recursive** attribútumot rendeljük. FORTRAN-ban a rekurziót explicit módon kell kezelni. E célból **amatch**-nek egy veremre van szüksége, amelyben hívási szintenként tároljuk a szükséges információt.

Jelen esetben a verem mélységét az egy mintában előforduló lezártak száma határozza meg, ezért célszerű **pat**-on belül a lezártakat reprezentáló bejegyzésekben annyi helyet hagyni, hogy egy-egy lezártra vonatkozó összes információ elférjen benne. A vermet kapcsolt listaként valósítottuk meg, ahol minden elem az előzőre mutat vissza, a szükséges kapcsolatokat a **getpat** és a hozzá tartozó rutinok építik fel a minta kódolásakor. A **pat**-ban minden lezárthoz a következő szerkezetű bejegyzés tartozik:

pat(i+0)	[típus]	lezárt esetén **CLOSURE**-t tartalmaz
pat(i+1)	**COUNT**	az illeszkedések ismétlődésének számlálója
pat(i+2)	**PREVCL**	a mintán belüli előző lezárt indexe
pat(i+3)	**START**	az input sorban az illeszkedés kezdetére mutató index

A **COUNT**, **PREVCL** és **START** szimbolikus konstansok a bejegyzésen belüli relatív címmel egyenlők, **CLOSIZE** pedig a bejegyzés hossza, ami ebben az esetben 4.

Valahányszor a mintában egy lezárt következik, az ennek megfelelő bejegyzés átkerül a verembe, annak aktuális szeleteként. Az ismétlendő mintáról feltesszük, hogy **pat**-ban a lezárthoz tartozó bejegyzés *után* áll, és így **pat** végignézésekor a lezártat jelző indikátor előbb fordul elő, mint maga a minta; ezt a sorrendet a mintát tároló tömb kialakításakor kell létrehozni. A mintát annyiszor illesztjük, ahányszor csak lehetséges, majd az ismétlésszámlálót és a szövegbeli kezdő indexet a verem aktuális szeletében tároljuk, és megkíséreljük a további illesztést a következő mintaelemmel.

Ha egy mintaelem illesztése sikertelen, és az utolsó lezárt illesztése még rövidíthető, az **amatch** ezt a mintaelemet egy illesztéssel rövidíti és újra próbálkozik. Ha ez már nem lehetséges, visszalépés történik az ezt megelőző lezárthoz, és ennek a rövidítésére kerül sor. Csak az összes lehetőség kimerítése után tekinti **amatch** az illesztést sikertelennek. Természetesen, ha az illesztés már az első lezárt előtt sikertelen, nincs lehetőség további kísérletezésre.

A végső, nem rekurzív **amatch** a következő:

```
# amatch (nem rekurzív) — illeszkedés keresése lin(from)-tól
    integer function amatch(lin, from, pat)
    character lin(MAXLINE), pat(MAXPAT)
    integer omatch, patsiz
    integer from, i, j, offset, stack

    stack = 0
    offset = from          # következő feldolgozatlan karakter
    for (j = 1; pat(j) ¬= EOS; j = j + patsiz(pat, j))
        if (pat(j) == CLOSURE) {   # egy lezártra utaló bejegyzés
            stack = j
            j = j + CLOSIZE          # a CLOSURE átlépése
            for (i = offset; lin(i) ¬= EOS; )  # illesszünk amennyit
                if (omatch(lin, i, pat, j) == NO)    # csak lehet
                    break
            pat(stack+COUNT) = i — offset
            pat(stack+START) = offset
            offset = i      # az illeszkedést meghiúsító karakter
            }
        else if (omatch(lin, offset, pat, j) == NO) {   # nem lezárt
            for ( ; stack > 0; stack = pat(stack+PREVCL))
                if (pat(stack+COUNT) > 0)
                    break
            if (stack <= 0) {       # a verem üres
                amatch = 0          # a sikertelenség jelzése
                return
                }
```

```
            pat(stack+COUNT) = pat(stack+COUNT) - 1
            j = stack + CLOSIZE
            offset = pat(stack+START) + pat(stack+COUNT)
            }
        # különben omatch sikeres volt
    amatch = offset
    return               # siker
    end
```

Most már a kapcsolódó **patsiz** és **omatch** rutinok is elkészíthetők:

```
# patsiz — pat (n)-nél kezdődő mintaelem hossza
    integer function patsiz(pat, n)
    character pat(MAXPAT)
    integer n

    if (pat(n) == CHAR)
        patsiz = 2
    else if (pat(n) == BOL | pat(n) == EOL | pat(n) == ANY)
        patsiz = 1
    else if (pat(n) == CCL | pat(n) == NCCL)
        patsiz = pat(n + 1) + 2
    else if (pat(n) == CLOSURE)       # elhagyható
        patsiz = CLOSIZE
    else
        call error("jelzés patsiz-ból: ez lehetetlenség")
    return
    end
```

A lezártak előfordulását ellenőrző sort „elhagyható"-nak minősítettük, mivel a **patsiz** rutint soha nem hívjuk ilyen minta hosszának meghatározására. Eredetileg, a tervezés kezdetén a biztonság kedvéért szerepeltettük, hátha változtatunk az adatszerkezeten; később mardosó bizonytalansági érzésünk enyhítése céljából hagytuk meg. Egy függvénynek azt kell tennie, *amit a neve mond*, még akkor is, ha adott esetben erre nincsen szükség, mivel előfordulhat, hogy egyszer csak egy programozó úgy módosítja a **find**-ot, hogy az valamely újfajta módon hívja a függvényt. A programozóknak joguk van figyelmen kívül hagyni programunk számos részletét, miközben számottevő módosításokat hajtanak végre rajta.

```
# omatch — egyetlen minta illesztési kísérlete pat(j)-nél
    integer function omatch(lin, i, pat, j)
    character lin(MAXLINE), pat(MAXPAT)
    integer locate
    integer bump, i, j

    omatch = NO
    if (lin(i) == EOS)
        return
    bump = -1
```

```
        if (pat(j) == CHAR) {
            if (lin(i) == pat(j + 1))
                bump = 1
            }
        else if (pat(j) == BOL) {
            if (i == 1)
                bump = 0
            }
        else if (pat(j) == ANY) {
            if (lin(i) ¬= NEWLINE)
                bump = 1
            }
        else if (pat(j) == EOL) {
            if (lin(i)==NEWLINE)
                bump = 0
            }
        else if (pat(j) == CCL) {
            if (locate(lin(i), pat, j + 1) == YES)
                bump = 1
            }
        else if (pat(j) == NCCL) {
            if (lin(i) ¬= NEWLINE & locate(lin(i), pat, j + 1) == NO)
                bump = 1
            }
        else
            call error("jelzés omatch-ből: ez lehetetlenség")
        if (bump >= 0) {
            i = i + bump
            omatch = YES
            }
        return
        end
```

A **bump** azt mutatja, hogy mennyivel kell előrelépnünk az inputban, ha **omatch** sikeresen illeszti a minta következő karakterét. Ez az üres sorozatokhoz illeszkedő minták esetén 0, egyébként 1.

A **locate** függvény egy karakterosztályon belül keres egy karaktert.

```
# locate — c keresése a karakterosztályban pat(offset)-nél
  integer function locate(c, pat, offset)
  character c, pat(MAXPAT)
  integer i, offset
  # az osztály mérete pat(offset)-ben van, ezt követik a karakterek
  for (i= offset + pat(offset); i > offset; i = i — 1)
      if (c == pat(i)) {
          locate = YES
          return
          }
  locate = NO
  return
  end
```

A **patsiz** és az **omatch** utolsó változatai érdeklődésre tarthatnak számot. Mivel a program maga építi fel a saját mintáit, pontosan tudjuk, hogy milyen fajta elemek fordulhatnak elő benne. Ha az **else if** lánc segítségével egy kivételével az összes lehetőséget kimerítettük, már nem szükséges ellenőriznünk, hogy a minta valóban megfelel-e az utolsó változatnak. Vagy mégis?

A **pat** tömb áttekinthetetlenül zavarossá vált. Valójában lineáris listára lenne szükségünk, melynek különféle típusú elemei vannak.

BOL, EOL, ANY	azonosító kód
CHAR	azonosító kód
	az összehasonlítandó karakter
CLOSURE	azonosító kód
	ismétlésszámláló
	előző lezártra mutató pointer
	az első illesztett karakterre mutató pointer
CCL, NCCL	azonosító kód
	az osztály mérete
	az osztály karaktereinek listája

Elvileg minden elemtípusnak tartalmaznia kellene egy mutatót a következő listaelemre. A FORTRAN-ban azonban nem tudunk tetszőleges struktúrákat leírni, mivel nem foglalhatunk le tárat egy, a nyelvből kezelt memóriabankból, és még mutatókról sem beszélhetünk. Ezért csalnunk kell. Különböző adattípusokat ábrázolunk egészekként, egy egész típusú tömbben alakítjuk ki saját tárfelosztásunkat, indexeket használunk mutatóként, de alkalmazásukat, amennyire csak lehet, elkerüljük, mivel eleve tudjuk, hogy mi hol található a tömbben. Az ilyen trükkök lehetővé teszik, hogy sok olyasmit megoldjunk FORTRAN-ban, amire ez a nyelv nincs felkészítve, ám ezért az olvashatóság és egyes érvényességi vizsgálatok hiánya az ár.*

Problémákra számíthatunk tehát, amelyekre fel kell készülnünk. Nagyon sok olyan általános programozási és tervezési hiba fordulhat elő, amely elrontja **pat** tartalmát. Ha vakon bízunk abban, hogy minden rendben van, **patsiz** és **omatch** esetleg értelmetlenséget fog megpróbálni feldolgozni ugyanúgy, mintha az valódi minta volna. A változatok sorozatában a bonyolultabbakat hagytuk a végére és **omatch**-on belül a negált karakterosztályok (**NCCL**) zárták le a sorozatot. Az

* A legjelentősebb hátrány a program függősége a választott adatstruktúrától: ha ezt megváltoztatjuk, a program ellenőrizhetetlen módon tönkremehet! *(A lektor)*

ilyen értelmetlenség önmagában is elég baj, de legalattomosabb fajtája az, amelyben egy számláló helyén valami más van. Ezért vizsgáljuk meg az utolsó lehetséges feltételt is és írjuk ki a „lehetetlenség" üzenetet, ha a minta nem értelmezhető.

Amikor a programot először futtattuk, azt mondta: „lehetetlenség". Aztán ezt az üzenetet még legalább százszor kaptuk meg, mire a program többi részét hozzákapcsoljuk és a részeket „összeráztuk". Ez a tapasztalat önmagáért beszél: ha egy magasan kifeszített kötélen akarunk egyensúlyozni, használjunk védőhálót. Elgondolkozhatunk azon, hogy mennyivel nehezebb lett volna egy olyan program belövése, amely elindítás után ellenőrizhetetlen módon elszabadul.

Még a kezdeti fejlesztési szakaszt követően is előfordult, hogy a **find** egyes részeit — néha alapvetően is — módosítottuk. Valahányszor egy módosítás során a lelkesedés az óvatosság rovására ment, a „lehetetlenség" üzenet terelt vissza minket a helyes útra. Végül is úgy döntöttünk, hogy az üzeneteket véglegesen meghagyjuk, és nem teszünk úgy, mintha a program tökéletes lenne. A hibaüzeneteknek a programból való eltávolítása „mivel az most már működik", ahhoz hasonló, mintha ejtőernyőt viselnénk a földön, de levennénk azt a felszállás után.

Az új programot alaposan ellenőrizhetjük olyan imitált **getpat** rutinok segítségével, amelyek előre meghatározott mintákat hoznak létre. Először ismét egyetlen „**a**"-val kísérletezünk, akárcsak korábban, de most már egy teljes működő **match** program áll rendelkezésünkre. Ezután az újabb és újabb lehetőségeket egyenként vizsgáljuk meg, míg végül eljutunk a többszörös lezártakat tartalmazó minták tetszőleges kombinációihoz.

Gyakorlatok

5.6 Becslés vagy mérés útján határozzuk meg, hogy mennyivel növekszik a végrehajtási idő, ha **patsiz**-ben és **omatch**-ben benne hagyjuk a kipróbáláskor használt programrészeket? A program teljes futási idejének hányadrészét teszi ez ki? Milyen kapcsolatban van ez a mindennapi gyakorlatban használt különböző minták előfordulásának gyakoriságával? Mennyi tárkapacitást igényel ez a többletprogram? Hányadrésze ez a teljes programméretnek? □

5.7 Írjuk meg és ellenőrizzük **match**-et és alárendelt moduljait különböző imitált rutinok segítségével, melyek a **getpat** útján adnak mintákat. Soroljunk föl növekvő bonyolultsági sorrendben tíz szövegmintát, amelyet kipróbálnánk. (*Útmutatás:* milyen mintákat kell írnunk ahhoz, hogy a program minden részét megmozgassuk?) Próbáljuk ki ezeket! □

5.3 A MINTÁK ELŐÁLLÍTÁSA

Most, hogy a mintakeresőnk legnagyobb része működik, térjünk át a minta beolvasására és kódolására. Annak ellenére, hogy a **find** mindig olyan mintákkal dolgozik, melyek az első pozíción kezdődnek és egy **EOS**-sel fejeződnek be, célszerű, ha általánosabb, olyan mintaelőállító rutint hozunk létre, amelyben a mintát tetszőleges elhatárolóval lehet befejezni és amely megmondja azt, hogy honnan folytathatjuk a vizsgálatot, ha erre szükség van. (Ezt a mintagenerátort részben elvi okokból, részben azért részesítjük előnyben, mert jól szolgálja a jelen és a következő fejezetben kitűzött célokat.) Ily módon **getpat** egy tartalom nélküli program, amely kapcsolatot teremt **find** és **makpat** között, amelyek a tényleges munkát végzik.

```
# getpat — a paraméter átalakítása mintává
  integer function getpat(arg, pat)
  character arg(MAXARG), pat(MAXPAT)
  integer makpat

  getpat = makpat(arg, 1, EOS, pat)
  return
  end
```

Miközben **makpat**-ot írtuk, rájöttünk, hogy módosításával megkönnyíthetjük **find** alkalmazását. Ha a % nem egy minta elején szerepel, elveszti speciális jelentőségét, és ugyanez történik a dollárjellel is, ha nem a minta végén áll, továbbá a csillaggal, ha a minta legelején fordul elő. Ez sok esetben feleslegessé teszi, hogy ezeket a jeleket felhívójel előzze meg, amikor tényleges előfordulásukat keressük. (Egy hibaüzenet szintén feleslegessé válik, aminek ugyancsak nagyon örültünk.) Minden csillagot, amely nem a sor elején fordul elő, ellenőrzünk abból a szempontból, hogy nem írja-e elő olyasvalaminek az ismétlését, amihez egy üres jelsorozat is illeszkedik, ugyanis a program többi része nem készült föl ennek a helyzetnek a kezelésére. Ebben az esetben a csillagot nem tekintjük tényleges karakternek, hanem a mintát kihagyjuk, és hibaüzenetet adunk. Ez tűnik a gyakorlatilag legbiztonságosabb megoldásnak.

Hangsúlyozzuk, hogy ezek a működési módok olyan *ad hoc* döntések eredményei, amelyeket a mintagenerátor előállítása során hoztunk. Amint ez gyakran előfordul, sok furcsa helyzetről kiderült, hogy kezelése nincs meghatározva és programozás közben kellett rá megoldást találnunk. Mi a specifikációnak a felhasználó számára legkézenfekvőbbnek tűnő kiegészítését választottuk.

Nyilvánvaló azonban, hogy túlságosan sok ilyen megoldás már káros. Célunk mindig az, hogy tiszta, egyértelmű, könnyen megjegyezhető funkcionális leírást adjunk, nem pedig az, hogy akármiféle ódon szabású rutinokat írjunk és majd megpróbáljuk elviselni őket. A túl sok kivétel és ad hoc megoldás oda vezethet, hogy a programoknak mind a belövése, mind a felhasználása nehézkessé válik. Mint mindig, most is az arany középutat kell megtalálnunk.

A következőkben bemutatjuk **makpat**-ot, amely a paraméterként adott mintát kódolt formára alakítja és elhelyezi a **pat** tömbben. Az egyszerű eseteket maga intézi el, a bonyolultabbakhoz szubrutinokat használ.

```
# makpat — mintakészítés arg(from)-ból, végjel: delim
  integer function makpat(arg, from, delim, pat)
  character esc
  character arg(MAXARG), delim, pat(MAXPAT)
  integer addset, getccl, stclos
  integer from, i, j, junk, lastcl, lastj, lj

  j = 1        # pat indexe
  lastj = 1
  lastcl = 0
  for (i = from; arg(i) ¬= delim & arg(i) ¬= EOS; i = i + 1) {
      lj = j
      if (arg(i) == ANY)
          junk = addset(ANY, pat, j, MAXPAT)
```

```
        else if (arg(i) == BOL & i == from)
            junk = addset(BOL, pat, j, MAXPAT)
        else if (arg(i) == EOL & arg(i + 1) == delim)
            junk = addset(EOL, pat, j, MAXPAT)
        else if (arg(i) == CCL) {
            if (getccl(arg, i, pat, j) == ERR)
                break
            }
        else if (arg(i) == CLOSURE & i > from) {
            lj = lastj
            if (pat(lj) == BOL | pat(lj)==EOL | pat(lj)==CLOSURE)
                break
            lastcl = stclos(pat, j, lastj, lastcl)
            }
        else {
            junk = addset(CHAR, pat, j, MAXPAT)
            junk = addset(esc(arg, i), pat, j, MAXPAT)
            }
        lastj = lj
        }
    if (arg(i) ¬= delim)          # korai befejezés
        makpat = ERR
    else if (addset(EOS, pat, j, MAXPAT) == NO)    # nincs hely
        makpat = ERR
    else
        makpat = i
    return
    end
```

A **pat** tömbbe az összes bejegyzést az **addset** hívása útján valósítjuk meg. Ez ugyanaz a rutin, amelyet a **translit**-ban használtunk az átütések ellenőrzésére, egy bejegyzés tárolására és a tárolási index módosítására. (Az **esc** szintén a **translit**-ból származik; a felhívójel kezelését végzi, ha ilyenre szükség van.)

Ahelyett, hogy **addset** minden hívásakor megvizsgálnánk, hogy van-e hely az új karakter számára, figyelmen kívül hagyjuk az ilyen irányú információt (oly módon, hogy a **junk**-ba irányítjuk, mint a 2. fejezetben), mivel **addset** sohasem ír a megadott határon túlra. Ha van hely a mintát lezáró **EOS** számára, minden rendben van, ha nincs, akkor valami hiba történt és a **makpat ERR**-t ad vissza.

A karakterosztályokat a **getccl** kódolja. A **getccl**-et úgy szerveztük, hogy az **filset**-et (és a hozzá tartozó almodulokat) használja a karakterosztályoknak megfelelő elemek létrehozására, akárcsak a **translit**-ban. Ily módon, sok új program írását megtakarítjuk, némi biztosítékunk van arra, hogy a program hibátlan és tudjuk, hogy a **find** és a **translit** azonos szabályok szerint határozza meg a karakterosztályokat.

```
# getccl — karakter oszt. kifejtése arg(i)-től pat(j)-be
  integer function getccl(arg, i, pat, j)
  character arg(MAXARG), pat(MAXPAT)
  integer addset
  integer i, j, jstart, junk

  i = i + 1          # a [ kihagyása
  if (arg(i) == NOT) {
       junk = addset(NCCL, pat, j, MAXPAT)
       i = i + 1
       }
  else
       junk = addset(CCL, pat, j, MAXPAT)
  jstart = j
  junk = addset(0, pat, j, MAXPAT)     # hely a számláló részére
  call filset(CCLEND, arg, i, pat, j, MAXPAT)
  pat(jstart) = j - jstart - 1
  if (arg(i) == CCLEND)
       getccl = OK
  else
       getccl = ERR
  return
  end
```

A **CCLEND** természetesen a **]**.

Mivel lezárt típusú elemek viszonylag ritka előfordulására számítunk, a **makpat** külön rutint, a **stclos**-t használja ezen elemek létrehozására és a hasonló elemekhez való kapcsolásra (azaz verem létrehozására). A visszaadott értéket **lastcl**-ben tároljuk, a lista későbbi felépítése céljából. Amikor *-gal találkozunk, az előző mintát annyival kell elmozdítani, hogy be tudjunk szúrni egy lezártat jelző elemet, hiszen **amatch**-nak ezt kell először látnia a működése során. Az **stclos** ezt is elintézi, miközben gondosan ügyel arra, nehogy a minta végét kitolja a minta végén túlra.

```
# stclos — lezártra von. bejegyzés elhelyezése pat(j)-nél
  integer function stclos(pat, j, lastj, lastcl)
  character pat(MAXPAT)
  integer addset
  integer j, jp, jt, junk, lastcl, lastj

  for (jp = j - 1; jp >= lastj; jp = jp - 1) {     # helycsinálás
       jt = jp + CLOSIZE
       junk = addset(pat(jp), pat, jt, MAXPAT)
       }
```

```
j = j + CLOSIZE
stclos = lastj
junk = addset(CLOSURE, pat, lastj, MAXPAT)  # a lezárt
                                            # elhelyezése
junk = addset(0, pat, lastj, MAXPAT)        # COUNT
junk = addset(lastcl, pat, lastj, MAXPAT)   # PREVCL
junk = addset(0, pat, lastj, MAXPAT)        # START
return
end
```

Amikor a **find**-ot fejlesztettük, a **pat**-ban tárolt azonosító kódok, amikor csak lehetett, a paraméterből vett tényleges karakterek voltak. (Ez talán nem volt teljesen nyilvánvaló, mivel olyan szimbolikus konstansokat használunk, mint az **EOL** és a **CLOSURE**, a **DOLLAR** és a **STAR** helyett, hogy szükség esetén az Olvasó maga választotta karakterekkel helyettesíthesse őket.) Hacsak egy mód van rá, nyomtatható belső kódokat használunk és így olyan utasításokat helyezhetünk el a belövés során, mint

```
call putlin(pat, ERROUT)
```

és ennek eredményeképpen többé-kevésbé olvasható kimenetet kapunk.Történetesen a **CHAR**-t **a**-ként és **NCCL**-t pedig **n**-ként definiáltuk. A számlálók és az indexek kinyomtatva furcsán festenek, de ennek ellenére a legtöbb minta jórészt olvasható.

A **find** felépítése során mindössze három tesztelési szintet említettünk, de valójában ennél sokkal több volt. Amint létrehoztuk valamelyik mintatípus kezelési mechanizmusát, azonnal beillesztettük és kipróbáltuk a már meglevő programvázzal. Mivel most egy kész tervet és terméket nyújtunk át az Olvasónak, mesterkélten hatna, ha bejárnánk azt a sok zsákutcát, melybe belekerültünk. Amikor az Olvasó a saját programját tervezi felülről lefelé haladva, minden bizonnyal több mint három teszttel kell hogy számoljon!

A lezártak kezelésének sínre állítása volt a legnehezebb, mivel ennek programja teszi ki a programnak mintegy felét. A legtöbb hibát, amint ez várható is volt, a **pat** tömbben követtük el. Vagy rosszul építettük föl a mintaelemeket, vagy nem úgy értelmeztük őket, ahogyan kellett volna. Mindkét esetben a program a „lehetetlenség" üzenettel reagált, amikor az új szolgáltatást kipróbáltuk. Ez megkönnyítette a hiba behatárolását és kijavítását.

Érdemes megemlíteni, hogy majdnem minden tervezési probléma a szimulált struktúrákból, dinamikus tárfelosztásból és rekurzióból származott. Jogosnak tekinthető az a kifogás, hogy ezen lehetőségek a FORTRAN-ból és sok más nyelvből hiányoznak. Ha olyan dolgok megvalósításával kell időt vesztegetnünk, melyeket a nyelv is elintézhetne helyettünk, ez elaprózza energiánkat és elfecsérli az eredeti cél megvalósítására szánt időnket. A tanulság megint világos: az, hogy aköznapi dolgokat „valahogyan" megoldani, kevés. Az is szükséges, hogy elintézésük kényelmes is legyen.

A **find** nem nagy program, mindössze mintegy 225 sor, ha nem számítjuk a **translit**-ból átvett részeket. Mégis elég nagy ahhoz, hogy elkészítsük a szubrutinok és függvények közötti kapcsolatot illusztráló diagramot, amint ezt a 3. Fejezetben szereplő **archive** esetében is tettük:

```
find
    getpat
        makpat
            getccl
                filset
            stclos
    match
        amatch
            omatch
                locate
            patsiz
```

Ebben a szerkezetben nem fordulnak elő a **getlin**, a **putlin** és az **addset**-hez hasonló segédrutinok, valamint a **filset** által hívott számos, a **translit**-hoz a 2. Fejezetben kifejlesztett rutin.

Figyeljük meg, hogyan halad a hierarchia fő ága, a **getpat** és a **match**, az absztrakttól a konkrét felé, amint a fa gyökerétől távolodunk. Ez természetes eredménye a **find** kidolgozási módszerének. Először egy absztrakt problémát tekintettünk, egy minta összes előfordulásának megkeresését az input szövegben és ezt két lépésben finomítottuk:

a minta átvétele
az input sorok összevetése a mintával

A továbbiakban mindkettőt tovább finomítottuk, mégpedig először a másodikkal foglalkoztunk, mindaddig, amíg a legalacsonyabb szinteken az összes részletet helyükre nem tettük. Minden egyes fázisban, ha az elvégzendő művelet eléggé egyszerű volt, akkor közvetlenül beprogramoztuk azt, ha pedig nem, akkor egy részét alacsonyabb szintű rutinok hívásainak formájában írtuk meg, melyet azután később dolgoztunk ki.

A felülről való elindulást és a részletek kidolgozásában való lefelé haladást gyakran „fokozatos finomítás"-nak nevezzük. Ez még szerény méretű programok esetén is jó megközelítés, mivel a tervezés sehol nem vész el részletekben, melyeket a finomítás későbbi fázisaira halasztunk. Az ellenőrzést korán el lehet kezdeni, mivel a kidolgozatlan részeket ideiglenes imitált rutinokkal tudjuk helyettesíteni, melyek csupán nagyon kevés számú funkciót látnak el. (Így jártunk el a **find** esetében.) A módosítások is könnyebben végezhetők el, mivel a megvalósítás különböző szempontjai általában különböző szinteken jelennek meg. A lényeg az, hogy minden egyes fázisban felismerjük a megfelelő absztrakciós szintet, és elkerüljük, hogy alacsonyabb szinthez tartozó részletekbe bonyolódjunk.

Gyakorlatok

5.8 Írjuk meg a **find** hiányzó rutinjait, és lőjük be azokat. Alkalmazzuk az előzőleg használt teszteket! Találunk-e újabbakat a **makpat** alaposabb kipróbálására? (Ismételten feltesszük a kérdést: végrehajtásra kerül-e a program *minden* része?) □

5.9 Most, hogy a **find** egy változata már működik, egészítsük ki több file felsorolásának lehetőségével úgy, hogy a

find *minta file1 file2 . . .*

parancs sorrendben dolgozza fel az adott file-okat, illetve ha file-paramétert nem adunk meg, a standard inputról olvasson. Ha több file-paraméter is van, minden

illesztett sor elé írassuk ki a file nevét. Előfordulhat-e, hogy ki akarjuk kapcsolni ezt a többlet-kiírást? □

5.10 Módosítsuk a **find**-ot úgy, hogy a

find ¬ *minta*

parancs nyomtassa ki mindazokat a sorokat, melyek nem illeszkednek a mintához. Melyik rutint kell evvel a vizsgálattal kibővíteni? Hogyan adnánk meg egy olyan mintát, amely a ¬ jelsorozat betű szerinti előfordulásával kezdődik? □

5.11 Bővítsük ki a **find**-ot a + metakarakterrel, amely egy minta egy- vagy többszöri előfordulását jelöli, akárcsak a csillag a zérus vagy többszöri előfordulást. □

5.12 Dolgozzuk ki egy tetszőleges bitsorozat mintában való megadásának a szintaxisát. Hajtsuk végre a **find**-nak olyan módosítását, amely ilyen minták feldolgozására is képessé teszi. (Lásd az **esc** tárgyalását a 2. Fejezetben.) □

5.4 NÉHÁNY MÉRÉSI EREDMÉNY

A **find**-ot egy RATFOR programot tartalmazó 225 soros és 3540 karakteres inputfile felhasználásával sok mintával futtattuk le. Az alábbiakban közlünk néhány, a Honeywell 6070 központi egységén mért időt:

minta:	**% (sor eleje)**	**1 jel (nincs meg)**	**3 betűs szó**	**?*x (x nincs meg)**
teljes idő:	**2,3 s**	**2,4 s**	**2,4 s**	**15,0 s**
getlin	**78,4%**	**76,5%**	**76, 8%**	**12,0%**
putlin	**19,5**	**0,1**	**2,0**	**0,0**
match	**0,3**	**5,2**	**4,5**	**0,8**
amatch	**0,6**	**10,0**	**9,2**	**38,1**
omatch	**0,5**	**7,3**	**6,6**	**33,6**
patsiz	**0,3**	**0,0**	**0,1**	**15,4**

Amint ez gyakran előfordul, egyszerű minták esetén a központi egység idejét javarészt B/K feldolgozás teszi ki. (Az általunk használt **getlin** mindössze egy FORTRAN **read** utasítást hív a karakterenkénti beolvasás céljából; a **putlin** pedig a **write**-ot használja. Így ezeknek a központi egységben töltött idejét nem tudtuk befolyásolni.) A **?*x** minta, ahol **x** nem szerepel az inputban, nehéz eset, mivel a **find**-nak minden sor minden egyes karakterén vissza kell lépnie, mielőtt eljutna ahhoz a következtetéshez, hogy a karakter nem fordul elő a sorban. A **?*?*x** jellegű minta természetesen még rosszabb lenne.

Annak eldöntésére, hogy **find** elég hatékony-e, a különböző mintákon mért teljesítményét súlyozni kell a minták előfordulásának gyakoriságával és a feldolgozandó anyag méretével. Ha az elvégzendő munka kizárólag kellemetlen mintákból és nagy szövegekből áll, akkor **find**-nak okvetlenül jobb algoritmust kell használnia. Jelenlegi formájában a **find** nem nagyon javítható, mivel az összes illesztésekre vonatkozó visszalépést ír elő, hogy biztosítsa a legbaloldalibb és leghosszabb illeszkedést. Lehetőség van azonban arra, hogy tetszőleges reguláris kifejezést visszalépés nélkül ismerjünk fel. A leghatékonyabb módszerek a reguláris kifejezést egy automatává alakítják át, amely egyidejűleg vizsgálja az összes lehetséges illeszkedést az input olvasása folyamán, és jelzi amikor valamelyiket megtalálta. Egy ilyen automata létrehozása természetesen időigényesebb kódolást igényel, mint az, amit mi használtunk, de

ennek megfelelően nagyobb megtakarítást ad futási időben. Részletesebben lásd a 9. Fejezetet a „The Design and Analysis of Computer Algorithms" című* könyvben, melyre a jelen fejezet végén hivatkozunk.

Szerencsére a bonyolult minták és a terjedelmes bemenetek találkozása a gyakorlatban nem sűrűn fordul elő. Leggyakrabban egy adott szót keresünk, és erre a **find** elfogadhatóan hatékony.

5.5 SZÖVEGVÁLTOZTATÁS

Most, hogy már képesek vagyunk mintákat azonosítani a szövegben, foglalkozhatunk egy szelektív változtatásokat végrehajtó hasznos eszközzel. A változtatás rendszerint három különböző művelet valamelyikét jelenti. Amikor elhagyjuk azt, amit megtaláltunk, *törlésről* beszélünk; amikor valami újat teszünk a helyébe, akkor *helyettesítést* végzünk, és amikor érintetlenül hagyjuk a megtalált mintát, de valamit elhelyezünk előtte vagy utána, akkor *beszúrást* hajtunk végre. Sok „felújítást" végző segédprogram alaposan felfújja ezeket a különbségeket, pedig a köztük levő eltérés lényegtelen.

Egy egyszerű jelölésmóddal ki tudjuk fejezni mindezeket a műveleteket is, valamint néhány további érdekes változatot is. A **change** program elfogadja a

change *from to*

parancsot, és ennek hatására megkeresi a *from*-hoz illeszkedő szövegminták összes előfordulását és mindegyiket a *to* helyettesítő sorozatra cseréli ki. A helyettesítő sorozat állhat egyszerűen a behelyettesítendő jelekből:

change mispell misspell

vagy lehet üres is, ha törölni akarunk:

change "very, "
change " *$"

Tartalmazhatja a speciális „szóismétlő" **&** jelet, amely visszahelyezi az illesztett darabot és ily módon beszúrást eredményez:

change active in&

melynek hatására az **active** minden előfordulása **inactive**-ra változik.

A szóismétlő jel bármelyik oldalon megjelenhet:

change able &-bodied

állhat középen is:

change a+b (&)

és többször is előfordulhat:

change very "&, &"

A felhívójel segítségével betű szerinti előfordulásban is használható:

change and @&

A **change** esetében nemcsak azt kell tudnunk, hogy melyik sorban fordul elő a mintához való illeszkedés, hanem azt is, hogy a sor melyik *részsorozata* eredményezett sikeres illesztést. Feladatunk az, hogy ebben a részsorozatban elvégezzük az előírt változtatásokat, majd további illeszkedéseket keressünk a sor hátralevő részében. A **find** a legbaloldalibb, leghosszabb illeszkedést találja meg. Ha ez az illesztés már megvan, a vizsgálatot az illesztett részsorozat után következő első bemenőjelnél folytathatjuk és ily módon megtaláljuk egy adott mintának a szöveg-file-ban való összes (átfedés nélküli) előfordulásait. Soha nem vizsgálunk újra behelyettesített szöveget, és így elkerüljük a végtelen ciklus kialakulásának lehetőségét.

A **change** megvalósításához csak kevéssel kell kiegészítenünk a **find** programját. Kezdettől fogva ügyeltünk annak pontos meghatározására, hogy a szövegrészek hogyan illeszkednek a mintát alkotó jelsorozatokhoz, így az illeszkedő sorozat hossza jól meghatározott volt. Az **amatch**-t úgy írtuk meg, hogy az illeszkedés végét követő első karakter indexét adja eredményül. A **change** felső szintjén levő program a következő:

```
# change — csere "from"-ról "to"-ra
    character lin(MAXLINE), new(MAXLINE), pat(MAXPAT),
              sub(MAXPAT)
    character arg(MAXARG)
    integer addset, amatch, getarg, getlin, getpat, getsub
    integer i, junk, k, lastm, m

    if (getarg(1, arg, MAXARG) == EOF)
        call error("hívás: change from to.")
    if (getpat(arg, pat) == ERR)
        call error("hibás from minta.")
    if (getarg(2, arg, MAXARG) == EOF)
        arg(1) = EOS
    if (getsub(arg, sub) == ERR)
        call error("hibás to.")
    while (getlin(lin, STDIN) ¬= EOF) {
        k = 1
        lastm = 0
        for (i = 1; lin(i) ¬= EOS;) {
            m = amatch(lin, i, pat)
            if (m > 0 & lastm ¬= m) {  # illesztett minta helyettesítése
                call catsub(lin, i, m, sub, new, k, MAXLINE)
                lastm = m
                }
            if (m == 0 | m == i) {  # nincs, vagy üres illeszkedés
                junk = addset(lin(i), new, k, MAXLINE)
                i = i + 1
                }
```

```
        else                    # az illesztett szöveg átlépése
            i = m
        }
    if (addset(EOS, new, k, MAXLINE) == NO) {
        k = MAXLINE
        junk = addset(EOS, new, k, MAXLINE)
        call remark("csonkított sor.")
        call putlin(new, ERROUT)
        call putch(NEWLINE, ERROUT)
        }
    call putlin(new, STDOUT)
    }
stop
end
```

A paraméterek elemzését, és megfelelő formára hozását a **find** számára készült **getpat**, valamint a **getsub** végzi, melyre rövidesen visszatérünk.

Minden sor kezdőpozíciójánál megkíséreljük az illesztést **amatch** segítségével. Ha az illesztés sikeres, **(m>0)** és nem azonos az utolsó illesztéssel **(lastm ¬= m)** a **catsub** összeállítja a **new** soron belül a kifejtett helyettesítési mintát a **k**-adik pozíciótól kezdve, és a teljes illesztett jelsorozatot átugorja. Ha az illesztés sikertelen **(m==0)** vagy üres jelsorozathoz történt **(m==i)**, akkor egy karakter átmásolódik a kimenetre és kimarad a bemenetről.

A fő problémát az jelenti, hogy mit kezdjünk az üres jelsorozatokhoz való illesztésekkel, mivel, hacsak nem vagyunk nagyon óvatosak, váratlanul üres sorozatokkal találkozhatunk. A **change**-t úgy építettük fel, hogy soha ne fordulhasson elő két szomszédos üres sorozat. Ez biztosítja azt, hogy az **a*** minta az **xy** sorhoz három ponton illeszkedjék: az **x** előtt, az **x** és **y** között, és az **y** után. Arra is gondunk volt, hogy az **a*** az **xay** sorhoz is csak három helyen illeszkedjék; ez még a legkevésbé meglepő viselkedés a **change** részéről.

A hibaellenőrzés jelenti a következő problémát, amely elég nagy terjedelmű programot igényel. Bár az eredménysorok túlcsordulása elkerülhető lenne oly módon, hogy a **putc**-t közvetlenül a **catsub**-ból hívjuk, ez tönkretenné egy olyan modul általánosságát, melyet később is használni akarunk. A figyelmeztető üzenet jobb megoldásnak tűnik.

A **getsub**, a **getpat**-hoz hasonlóan, a helyettesítő minta felépítését két részre osztja: egy specifikusra és egy általánosabbra.

```
# getsub — helyettesítő minta átvétele sub-ba
    integer function getsub(arg, sub)
    character arg(MAXARG), sub(MAXPAT)
    integer maksub

    getsub = maksub(arg, 1, EOS, sub)
    return
    end
```

A **maksub** átmásolja a helyettesítő mintát **sub**-ba, egészen az elhatárolójel, jelen esetben egy **EOS** előfordulásáig. Az & szóismétlő jel minden előfordulása speciális kóddal helyettesítődik.

```
# maksub — helyettesítő sorozat tárolása sub-ban
   integer function maksub(arg, from, delim, sub)
   character esc
   character arg(MAXARG), delim, sub(MAXPAT)
   integer addset
   integer from, i, j, junk

   j = 1
   for (i = from; arg(i) ¬= delim & arg(i) ¬= EOS; i = i + 1)
        if (arg(i) == AND)
             junk = addset(DITTO, sub, j, MAXPAT)
        else
             junk = addset(esc(arg, i), sub, j, MAXPAT)
   if (arg(i) ¬= delim) # végjel hiányzik
        maksub = ERR
   else if (addset(EOS, sub, j, MAXPAT) == NO)   # nincs hely
        maksub = ERR
   else
        maksub = i
   return
   end
```

A **DITTO** minden ábrázolható jeltől megkülönböztethető kód, akárcsak az **EOF** és az **EOS**; legcélszerűbb egy kicsiny, negatív értékkel kifejezni.

Csupán a **catsub** van hátra, amely egyszerű:

```
# catsub — helyettesítő szöveg csatolása "new" végéhez
   subroutine catsub(lin, from, to, sub, new, k, maxnew)
   integer addset
   integer from, i, j, junk, k, maxnew, to
   character lin(MAXLINE), new(maxnew), sub(MAXPAT)

   for (i = 1; sub(i) ¬= EOS; i = i + 1)
        if (sub(i) == DITTO)
             for (j = from; j < to; j = j + 1)
                  junk = addset(lin(j), new, k, maxnew)
        else
             junk = addset(sub(i), new, k, maxnew)
   return
   end
```

Gyakorlatok

5.13 Hogyan változtatja meg a

change aktiv in&

parancs az *inaktív*, az *attraktív* és a *radioaktív* szavakat? Milyen eljárást alkalmaznánk, ha meg akarnánk győződni arról, hogy egy szövegen végrehajtott „kis" változtatások valóban a kívánt eredménnyel jártak-e? □

5.14 Mi történik, ha valamit újsor-karakterré akarunk változtatni? Mi történik, ha megpróbáljuk eltávolítani a sor végéről az újsor-karaktert? □

5.15 Van-e olyan feladat, amit a **translit**-tel meg tudunk oldani, de a **change** rutinnal nem? □

5.16 Bővítsük ki a **change**-t többszörös módosítások végrehajtásával. Például a

change a b c d

parancs minden **a**-t **b**-re, majd minden **c**-t **d**-re változtat az eredménysorozatban. Egyenértékű-e ez a

change a b | change c d

csatlakozással minden lehetséges minta és helyettesítési sorozat esetén? □

5.17 Tekintsünk egy olyan file-t, melyben minden sor két, egymástól egy tabulátorkarakterrel elválasztott mezőből áll. Programcsatlakozás segítségével hozzuk létre azt az új file-t, amelyben a két mező minden sorban fel van cserélve, pl. a

1234 5678

sorból legyen

5678 1234

(Útmutatás: Próbáljuk meg minden sor tartalmát duplázni, egy duplikátok közötti elválasztójellel.) □

*

Gyakran hasznos, ha meg tudjuk jelölni egy szövegminta részeit abból a célból, hogy egy illesztett jelsorozat darabjait szelektíven visszahelyezhessük vagy átrendezhessük. Tegyük föl, hogy bevezetünk két metakaraktert, a { és } kapcsos zárójeleket, melyeknek egy szövegmintában való előfordulása a minta megfelelő részéhez illeszkedő részsorozat megjegyzésére szólít fel. Például, a következő mintában

%{???} {?*}

az első zárójelpár a sor elején levő három jel tárolását írja elő, bármik legyenek is azok; a második zárójelpár pedig a sor többi részének tárolását. Szükségünk van még a megjegyzett részsorozatokat előhívó jelölésre. Tegyük föl, hogy @**n** az *n*-edik zárójelpár által megjegyzett jelsorozatra utal, ahol **n** egyetlen számjegy. Ennek segítségével a következőképpen tudunk egy háromkarakteres sorszámot a sor elejéről a végére átvinni:

change %{???}{?*} @**2**@**1**

Nehezebb példaként tekintsük a következőt:

change {[¬@**t**]*}@**t**{?*} @**2**@**t**@**1**

amely felcserél két, egymástól tabulátorral elválasztott mezőt, amint ezt az előző gyakorlatban meglehetősen „mezítlábas" módszerrel végeztünk el.

*

5.18 Írjuk át a mintakereső programot oly módon, hogy megjegyezze a megjelölt mintákat, majd módosítsuk a **change**-t úgy, hogy kívánságra beillessze ezeket a helyettesítő sorozatba. Tud-e a programunk egymásba skatulyázott zárójelpárokat kezelni? Fontolóra vehetjük a @**(** és @**)** használatát { és } helyett, különösen, ha gépünk jelkészlete korlátozott? Mennyiben jobb ez a konvenció, mintha a közönséges zárójeleket metakarakterekké nyilvánítanánk? □

5.19 A mintamegjelölési lehetőség felhasználásával hogyan írnánk le azt, hogy egy sorban csak a legbaloldalibb számot kell nullára változtatni? Csak a jobbszélsőt? A balról számított másodikat? □

5.20 Egészítsük ki a **find**-ot a — **n** funkcióval, amely a sor elé kinyomtatja minden illesztett sor sorszámát, attól egy szóközzel elválasztva. Ez hasznos az illesztett sorok későbbi azonosítása céljából. Mi a hatása a

find —n % | sort —n —r | change "%[0—9]* "

parancsnak? (Gondoljunk arra, hogy **sort —n —r** a sor elején levő numerikus mező szerinti fordított rendezést végez.) Mennyire lenne nehéz egy egyedi programot írni ennek a feladatnak az elvégzésére, tetszőleges file méret esetén? Megérné-e a munkát? □

5.12 Egy D.E.KNUTH által felvetett probléma olyan nyolcbetűs szavak legbővebb halmazának megkeresése, melynek négy *középső* betűje azonos. Tegyük fel, hogy van egy géppel olvasható szótárunk (soronként egy szóval), hogyan oldanánk meg ezt a feladatot a **find, change, sort** és az **unique** felhasználásával? Szükségünk van-e más programra is? □

5.6 ÖSSZEFOGLALÁS

A szövegminták kezelésére számos eljárást és ezekhez tartozó jelölésrendszert vezettünk be. A következőkben röviden összefoglaljuk, hogy mit tudunk leírni ezek segítségével.

Egy szövegminta a következő elemekből áll:

c	karakter
?	bármelyik karakter, az újsor kivételével
%	a sor kezdete
$	a sor vége (vagy üres jelsorozat az újsor-karakter előtt)
[...]	karakterosztály (karakterek bármelyike)
[¬...]	negált karakterosztály (bármi, az adott karakterek kivételével)
*	lezárt (az előző minta zérus, vagy többszöri előfordulása)
@*c*	felhívójel után alkalmazott karakter (pl. @%, @[, @*)

A karakterek egyedi jelentése a szövegmintában megszűnik a [...]-en belül, ha a felhívójellel együtt fordulnak elő, és a következő esetekben:

%	nem a minta elején áll
$	nem a minta végén áll
*	a minta elején áll

Egy karakterosztály a következő elemekből áll, azaz ezekből zérust vagy többet tartalmazhat [és] között:

c	karakter, beleértve a [-t is,
a—c	karakterintervallum (számjegyek, valamint kis- és nagybetűk egyaránt előfordulhatnak)
¬	negált karakterosztály a minta elején
@*c*	felhívójellel együtt használt karakter (@ ¬ @— @@ @])

Egy karakterosztályban a karakterek speciális jelentése elvész, ha a felhívójellel együtt szerepelnek, vagy a következő esetekben:

¬	nem az osztály elején áll
—	az osztály elején vagy végén áll

Egy helyettesítő minta a következő elemekből áll, vagyis ezekből zérus vagy több jelet tartalmazhat:

c	tényleges karakter
&	szóismétlő, azaz bármi, ami illeszkedett
@*c*	felhívójellel együtt használt karakter (@ **&**)

A felhívójeles szerkezet a @ karakterből áll, melyet egyetlen karakter követ:

@**n**	új sor
@**t**	tabulátor
@*c*	c (beleértve @@-t)

BIBLIOGRÁFIAI MEGJEGYZÉSEK

A szövegminták tágabb osztályának felismerését átfogó módon tárgyaló anyag található A. V. AHO, J. E. HOPCROFT és J. D. ULLMAN: *The Design and Analysis of Computer Algorithms* (Addison—Wesley, 1974) című könyvének 9. fejezetében. Érdemes elolvasni J. F. GIMPEL *„A theory of discrete patterns and their implementation in Snobol4"* című, a CACM-ben 1973 februárjában megjelent cikkét is. A Snobol természetesen egy általánosan használt mintaillesztő nyelv. Kulcsszavak párhuzamos keresésére való gyors algoritmus leírását tartalmazza A. V. AHO és M. J. CORASICK *„Efficient string matching: an aid to bibliographic search"* című cikke (CACM, 1975. június).

A rekurzió egy kitűnő tárgyalása — mikor használjuk, és hogyan kerüljük el — található D. W. BARRON: *„Recursive Techniques in Programming"* című művében (Americal-Elsevier, 1968).

A **find**-ot a UNIX-ban szereplő **grep** segédprogram mintájára készítettük, mely K. L. THOMPSON munkája.

6. fejezet

Szerkesztés

Most, hogy már rendelkezésünkre áll a szövegminták illesztésére és cseréjére alkalmas program, nekigyürkőzhetünk a szövegszerkesztés — ezeknél általánosabb — problémájának megoldásához. A szövegszerkesztés szöveges információ (programok, adatok, dokumentumok stb.) létrehozását és módosítását jelenti. Minden interaktív (és a legtöbb kötegelt üzemmódú) rendszerben van valamilyen szerkesztési lehetőség, de ez gyakran kezdetleges. Sok felhasználó csak azért nem panaszkodik, mert nem tudja, hogy mit veszít. A szövegkörnyezetben minták alapján való keresés, globális cserék végrehajtása vagy a tetszőleges file-ra vonatkozó be- és kivitel gyakran még a „korszerű" szerkesztőprogramokból is hiányzik. Vannak olyan szerkesztők, melyek tartalmazzák ezeket a lehetőségeket, de parancs-szintaxisuk olyan bonyolult, hogy ez nagymértékben korlátozza használhatóságukat.

Az itt bemutatandó szerkesztőprogramot egy széles körben használt népes párbeszédes szövegszerkesztő-család utolsó tagjának mintájára terveztük. A terve az emberi szemléletmódra való törekvés hatja át, igyekszik tömör, szabályos és hatékony lenni. Mivel az **edit** elsősorban interaktív felhasználásra készült, ezért „testreszabott" és tömör, ugyanakkor könnyen használható. Szövegszerkesztő-program esetében ez különösen fontos, mivel a legtöbb felhasználó számára ez a rendszerrel való kapcsolattartás legfőbb eszköze. (A mi UNIX rendszerünkben a szerkesztés a végrehajtott parancsok 15%-át teszi ki, több mint háromszorosát az utána következő parancs részarányának.) Az **edit** azonban *nem csupán* párbeszédes szerkesztésre használható. Elindítható előkészített forgatókönyvből és más programokból is. Gyakran használják programok eredményeinek kiválogatására vagy további programok adatainak előkészítésére. Az **edit** tehát eszköz.

Ugyancsak fontos szempontot jelentett a szerkesztő tervezésében a hibakezelés. A 3. fejezetben szereplő **archive**-hoz hasonlóan, az **edit** is értékes file-okat kezel, ezért óvatosnak kell lennie. Továbbá, az interaktív használat közben a szerkesztő „nem dobhatja be a törülközőt", amikor a felhasználótól egy hibás utasítást kap. A hibakezelésben elágazásnak kell lennie, különben egy egyszerű elütés értékes információ elvesztését okozhatja.

Végül, mivel a szerkesztő természetszerűleg nagy program, jól kell megszervezni, mert különben teljesen áttekinthetetlen lesz (és ezért valószínűleg nem is tudja megvalósítani az emberi szemléletmóddal és a megbízhatósággal kapcsolatos célkitűzéseket sem). Ennek megfelelően, a szerkesztőprogramot felülről lefelé tervezzük, és a file-ok kezelésével, valamint a szöveg ábrázolásával kapcsolatos döntéseket a lehető legalacsonyabb szintre visszük le. Amennyire ez lehetséges, a megvalósítás részleteit elkülönítjük azon rutinoktól, melyeknek nem kell azt ismerniük, és így ezeket megváltoztathatjuk és tökéletesíthetjük a program nagy részének megzavarása nélkül.

6.1 AMIT A FELHASZNÁLÓ LÁT

Bár általában célszerűbb kicsiben kezdeni, és fokozatosan tovább fejlődni, egy szerkesztő, a programozási nyelvekhez hasonlóan, olyan gyakran használt program, hogy valóban jó minőségűnek kell lennie, ha a felhasználó nem akarja idejének nagy részét a hiányosságok leküzdésére pazarolni. Ennek megfelelően, az **edit** viszonylag sok lehetőséget kínál, a minimálisan szükségesnél jóval többet.

Ebben a részben röviden áttekintjük az **edit**-et és ennek során képet alkothatunk arról, hogy milyen döntéseket hoztak a tervezők és miért, és hogy milyen parancsok állnak rendelkezésünkre. Az egyes parancsok részletes magyarázatát a megvalósítás során adjuk.

A szerkesztés elindításához azt kell írni, hogy

edit

vagy

edit *file*

Az utóbbi esetben, ha a *file* már létezik, a program feltételezi, hogy a felhasználó ennek tartalmával akar dolgozni, ezért a file-t egy belső pufferba másolja át, mely utóbbinak a megvalósítását pillanatnyilag nem írjuk le közelebbről. Mindenképpen a pufferban levő szöveg módosul és később esetleg valamilyen külső file-ba kerül. File-okat csakis explicit paranccsal lehet módosítani. Ez biztonságosabb, mintha a file-on helyben végeznénk módosításokat, mivel egy elhibázott szerkesztés esetén még mindig beolvashatunk egy újabb másolatot és elölről kezdhetjük a munkát.

Az **edit** alapvetően „sorszervezésű", mivel a legtöbb szerkesztőparancs a puffer egy vagy több során operál. Ez természetes szervezés, mivel a bemenő szöveg, lényegéből folyóan, sorokra tördelve olvasódik be. Más egységeket is lehetne választani — karaktereket, szavakat, mondatokat vagy tetszőleges jelsorozatokat —, de az alkalmazások széles skáláján a sorok látszanak a legmegfelelőbbnek. Természetesen a sorok egyes részeihez is hozzá lehet férni, erre rövidesen visszatérünk.

Hangsúlyoznunk kell, hogy a szerkesztő semmilyen sorokon belüli szerkezetet nem tételez fel. Nem tudja pl., hogy a 6. és 73. oszlopnak egy FORTRAN program esetén különös jelentősége van, és nem ismer más speciális formátumot sem. Tapasztalatunk szerint azok a szerkesztőprogramok, melyek azt „hiszik", hogy túl sokat tudnak arról, amit a felhasználó csinálni akar, inkább akadályozzák, mint segítik a munkát.

Mint már említettük, az **edit** igyekszik tömör és szabályos lenni. Minden szerkesztőparancs egyetlen betűből áll. Ez előtt esetleg egy vagy két „sorszám" állhat, melyek a puffer azon sorainak zárt intervallumát mutatják, ahol a parancs működni fog. Így az

1p

parancs az első sor kinyomtatását jelenti, míg az

1,3p

az 1., 2. és 3. sort nyomtatja ki. Soronként csak egyetlen parancs megengedett, mivel ezzel csökken annak a lehetősége, hogy egy hibás input komoly kárt okoz.

A **d** törlési parancs hasonló a **p**-hez, és törli az adott intervallumban levő sorokat:

1,3d

törli a puffer első három sorát. Nemlétező sorra való hivatkozás mindig hibás; ilyen esetben az **edit** „panaszkodik".

A sorokat a puffer *kezdetéhez viszonyítva* számozzuk. Az első három sor törlése után az első megmaradó sor (a régi negyedik sor) lesz az új első sor, és az összes többi sor sorszáma ennek megfelelően módosul. Ez a viselkedés szokatlan lehet azok számára, akik olyan szerkesztőt szoktak meg, ahol a „sorszámok" a szövegsorok részeként fizikailag is léteznek. A mi sorszámaink nem képezik a file részét, és nincs semmiféle fizikai reprezentációjuk, csupán a soroknak a pufferen belüli relatív elhelyezkedését fejezik ki. Amint rövidesen megmutatjuk, ez a megoldás felbecsülhetetlen értékű rugalmasságot biztosít a sorok specifikálására és átrendezése esetén.

Bár a teljes szerkesztés elvégezhető akár relatív, akár abszolút sorszámok használatával, ez gyakran szükségtelen nyűgöt jelent, ezért az **edit** megengedi sorok specifikálásának több más módját is. Például, a szerkesztő folyamatosan megjegyzi az *aktuális sort,* ami általában az a sor, melyet az előző paranccsal dolgoztunk fel. Az aktuális sorra, a „." (pont) szimbólummal hivatkozhatunk, amely mindenütt használható, ahol egész számmal kifejezett sorszám állhat. A puffer *utolsó sora* szintén megadható; ezt $ jelöli. Ily módon a

```
.,$p
```

parancs kinyomtatja az aktuális sort, és a puffer végéig terjedő esetleges további sorokat; az

```
1, $p
```

mindent kinyomtat és az

```
1, $d
```

parancs mindent töröl.

A pont értékét több parancs is megváltoztatja. Így a **p** parancs után a pont az utolsó kinyomtatott sorra hivatkozik, a **d** parancs után pedig az első, nem törölt sorra, azzal a megkötéssel, hogy soha nem lépheti át a $-t. Ezért egy önmagában álló

```
d
```

törli az aktuális sort és a pontot a következő sorra állítja, a

```
.,$d
 p
```

pedig mindent töröl a puffer végéig, és kiírja az új utolsó sort.

A pont és a **$** azt a célt szolgálja, hogy ritkábban legyen szükség konkrét sorszámok használatára. Ugyancsak ezt támogatja a sorszámokra vonatkozó aritmetika is. Egy file utolsó néhány sorának kinyomtatására (pl. hogy megnézzük, mennyire jutottunk az előző szerkesztési menetben) a

```
$—10, $p
```

parancs használható, vagy azt is írhatjuk, hogy

```
.—5,.+5p
```

ha a pillanatnyilag aktuális sor környékéről, ahol előzőleg dolgoztunk, néhány sort akarunk kiírni.

A sorszámok alapján működő szerkesztés még a ponttal, dollárral és aritmetikával támogatva is nehézkes. Szerkesztés közben azt szeretnénk, ha lehetőségünk lenne a szerkesztőnek a „Keresd meg ennek és ennek a karakterláncnak egy előfordulását" utasítást adni, mert akkor ezzel dolgozhatnánk tovább anélkül, hogy pontosan tud-

nánk, hol is fordul elő. Az **edit** segítségével szövegkörnyezet alapján való kereséssel találhatunk egy-egy sort, ha egy mintát egyszerűen két törtvonal közé teszünk.

/abc/

ami azt jelenti, hogy: Az aktuális sort követő sorban indulva előrehaladó vizsgálatot kell végrehajtani mindaddig, ameddig olyan sort nem találunk, amelyik az **abc** mintához illeszkedik. A minta természetesen bármi lehet, amit az 5. fejezetben leírtunk; ennek biztosítására ugyanazt a mintaillesztő-programot fogjuk használni. A keresés a **$** sor után, ha szükséges, az első sornál folytatódik. Így a

/abc/, $p

parancs megkeresi a következő sort (az aktuális sor után), amely **abc**-hez illeszkedik, és innen kezdve kinyomja a puffer tartalmát. (Ha a keresés az utolsó sor után a puffer elején tovább folytatódik, majd eléri az aktuális sort anélkül, hogy az illesztés megtörtént volna, hibajelzés következik.)

Hasonlóképpen, visszafelé is indíthatjuk a keresést, ha egy mintát balra dőlő törtvonalak közé teszünk. Pl.

def\\

a következőket jelenti: Az aktuális sort megelőző sorban indulva visszafelé haladó vizsgálatot kell végrehajtani mindaddig, ameddig olyan sort nem találunk, amely a **def**-hez illeszkedik. A keresés ebben az esetben az első sor után az utolsó sorral folytatódik, és hibajelzést kapunk, ha egyetlen illeszkedő sor sincs a pufferban. Egy RATFOR program szerkesztésekor pl. azt mondhatjuk, hogy

subroutinel\\, **/end/p**

ha ki akarjuk nyomtatni azt a szubrutint, amely az aktuális sort tartalmazza.

Egy önmagában álló sorszám (melyet csak egy újsor-karakter követ), ezen sor nyomtatását előíró parancsként értelmeződik. Ennek megfelelően

$

kinyomtatja az utolsó sort. Az a gyakori tevékenység, hogy „Keresd meg a legközelebbi sort, amely az **abc**-t tartalmazza", az

/abc/

paranccsal írható elő. Ez megkeresi a sort, kinyomtatja, a pont értékét erre a sorra állítja, és így a munkát innen folytathatjuk. Speciálisan, egy önmagában álló újsor-karakter a következő sor kinyomtatását eredményezi, hogy soronként könnyen végigmehessünk a pufferen.

A szövegkörnyezet alapján való keresés igen fontos. Legtöbb esetben annak a segítségével jutunk el arra a következő helyre, ahol valamilyen szerkesztést akarunk végrehajtani. Gyakran még a forrássorok sorszámainak ismeretében is jobb szöveg szerinti keresést végezni. Ha például egy azonosítót kétféle módon használtunk, a lista javítása során egy vagy több előfordulása könnyen elkerülheti figyelmünket. A szövegkörnyezet alapján való keresés azonban a forrásprogram minden helyére elvezet, ahol a keresett azonosító előfordul.

A sorszámoknak a parancsok előtt (és nem utánuk) való használata első pillanatra természetellenesnek tűnik, de gyorsan megszokható. Ennek segítségével elérhető, hogy az egyes parancsok a parancs betűje után eltérő szintaxist használhassanak a nem kötelező információ megadására anélkül, hogy a sorszámok intervallumának egységes leírását megsértenénk.

A további információt igénylő parancsok közül a legfontosabb az **s** helyettesítési parancs, amely karakterek kicserélésére használható egy soron belül. Az

s/ofrmat/format/

parancs az aktuális sorban **ofrmat** *első* előfordulását *format*-ra cseréli. Ha a sorban egynél több **ofrmat** fordulna elő (ami remélhetőleg nem áll fenn), azt mondhatjuk, hogy

s/ofrmat/format/g

ami a cserét az egész sorban *globálisan* végrehajtja. (Természetesen az **s** parancs bal oldala bármilyen szabályos minta lehet, mivel ugyanaz a mintaillesztő-program működik a szövegkörnyezet alapján való keresés és a helyettesítés esetében is. A jobb oldal tartalmazhatja az & szóhelyettesítő jelet bárminek a rövidítésére, melyhez a bal oldal illeszkedett, ugyanúgy, mint az 5. fejezetben szereplő **change** programban.

A **g**-vel vagy **g** nélkül álló **s** parancsot egy **p** parancs követheti az utolsó érintett sor kinyomtatása céljából, annak ellenőrzésére, hogy a kívánt helyettesítés tényleg végrehajtódott-e. Az egyes parancsok után a nyomtatás nem automatikus, ezért az **edit** csak annyira bőbeszédű, amennyire a felhasználó azt igényli. (Egy törlési parancs után is írhatunk **p**-t, ilyenkor az első, nem törölt sor kerül kinyomtatásra.) Egy **s** parancsot egy vagy két sorszám előzhet meg annak jelzésére, hogy a helyettesítést több, egymás utáni sorban kell végrehajtani:

.+1s/ofrmat/format/

a hibát a következő sorban javítja, míg a

1,$s/ofrmat/format/g

a javítást minden sorban, akárhol elvégzi. (Ez igen hasznos következetes elírások esetén.) A pont az utolsó megváltoztatott sorra fog hivatkozni.

Az **s** valószínűleg a szerkesztőprogram leghasznosabb parancsa, mivel lehetővé teszi, hogy tömör formában fejezzük ki az egy vagy több sorra vonatkozó helyettesítéseket. Gyakran használják egy sor végéhez csatlakozó szöveg beszúrására:

s/$/új befejezés/

vagy egy szövegnek egy sor elején való elhelyezésére:

s/s%/új sorkezdet/

illetve a sor közepén való beillesztésre:

s/és/& továbbá/p

A helyettesítési utasítás egyes részeit elválasztó karakter nemcsak a törtvonal lehet, hanem bármi más is. Így pl.

s: / ::g

egy sorból törli az összes törtvonalat. Ezt természetesen elérhettük volna a felhívójelnek a törtvonalra való alkalmazásával is:

s/ @/// g

Ez azonban zavaró lehet.

A szövegkörnyezet alapján történő keresésben használt utolsó minta megőrződik és később a // vagy \\ üres minta segítségével lehet rá hivatkozni. Ha a **/format/** mintát használjuk egy **format** utasítás megkeresésére és nem azt találjuk meg, amelyiket kerestük, a // leírásával elmehetünk a következőig, illetve a \\ segítségével az

előzőig. A megőrzött minta sok munkaigényes és hibalehetőséget tartalmazó újraírástól kímél meg minket. Egy tipikus használata, helyettesítéssel együtt a következő:

```
/ofrmat/s//format/p
//s//format/p
//s//format/p
...
```

Ennek segítségével végigaraszolunk a szövegen, és egyenként javítjuk az elírásokat. Amikor a keresés először vált sikertelenné, akkor az összes helyettesítést elvégeztük. Valamennyi helyettesítést elvégezhettük volna az

```
1,$s/ofrmat/format/gp
```

paranccsal, de ez csak az utolsót nyomtatta volna ki ellenőrzés céljából.

Most visszatérünk azokra a műveletekre, amelyek teljes szövegsorokat érintenek. A legfontosabb a *hozzáillesztő utasítás*, amely új szövegsorokat helyez el a pufferben. Ez alapvető eszköze egy file további szöveggel való kiegészítésének, illetve egy file létrehozásának. Ennek a könyvnek a teljes szövegét, valamint a benne szereplő összes programját valamikor az **append** parancs segítségével illesztettük egy az itt bemutatandókhoz igen hasonló szövegszerkesztő pufferjához.

Gyakori alkalmazására való tekintettel, a hozzáillesztő utasítás a lehető legkényelmesebb. Amikor az **a** parancs előfordul, a szerkesztő egy speciális, *hozzáillesztő üzemmódba* megy át, amelyben minden bejövő szöveg a puffer megfelelő helyén tárolódik. A felhívójelek és egyéb speciális jelek elvesztik különleges jelentésüket mindaddig, amíg be nem jön egy egyetlen ponttal kezdődő sor. Ez a jelzés, amely könnyen írható és rendkívül valószínűtlen, hogy közönséges szövegben ebben a formában forduljon elő, jelöli a hozzáillesztő mód végét, és ő maga már nem íródik be a pufferba. Az ezt követő sorok újra parancsként értelmeződnek.

Ennek megfelelően, ha szöveget akarunk hozzáilleszteni a pufferhez, meg kell adnunk, hogy hol akarjuk azt elhelyezni, és végre kell hajtanunk egy hozzáillesztő utasítást. Például, a szövegnek a puffer végéhez való csatolása a következő módon történhet:

$a

bármit írhatunk ide
mit játszi elménk kitalál
lezárva egy különleges sorral
mely egyetlen .-ból áll

.

Ez négy sorral egészíti ki a puffert, majd folytatja a további parancsok feldolgozását. A negyedik sorban szereplő pontnál nem kell felhívójelet alkalmazni, mivel a pontnak csak akkor van különleges szerepe, ha egyedül áll a sor elején. Ha a puffer elejéhez akarunk hozzátenni valamit, a nullás sorszámot használhatjuk: pl. **0a**. Ha sorszámot nem adunk meg, a szöveg az aktuális sor után lesz beillesztve.

Az **i** *(insert)*, azaz beszúró-utasítás, hasonló az **a**-hoz, csak a sorok a megnevezett sor *elé* és nem *utána* kerülnek. A **c** *(change)*, azaz cserélő-utasítás egy vagy több sort nulla, vagy több sorral helyettesít; a

sor1, sor2 **c**
akármi

.

parancs a *sor1*-től *sor2*-ig terjedő szövegrész helyébe a **c** után álló szövegrészt helyettesíti. Ha sorszámot nem adunk meg, akkor mind az **i**, mind a **c** parancs az aktuális sorral dolgozik.

Nyilvánvaló, hogy az **a** és a **d** parancs mellett nincs szükség **c**-re vagy **i**-re. Az általuk nyújtott felhasználásbeli kényelem azonban megéri az egyébként nem jelentős programozási többletet.

A pont a pillanatnyi sor jele az utolsó hozzátett, kicserélt vagy beszúrt sorra mutat, így a szerkesztés közben elkövetett hibákat azonnal javítani tudjuk, amint ezt a következő sorozat mutatja. (A parancsokat dőltbetűs magyarázattal kísérjük.)

a	*szöveget akarunk hozzáilleszteni*
10 ofrmat (...)	*puff! egy hiba*
.	*a hozzáillesztés befejezése*
s/ofr/for/	*a javítás*
a	*visszatérés a hozzáillesztéshez*
	a javított sor után
... további szöveg...	
egy elrontott sor	*puff! ismét egy hiba*
.	*a hozzáillesztés befejezése*
c	*egyszerűen kicseréljük az egészet*
... javított szöveg...	*és folytatjuk a gépelést*

A pillanatnyi és az alapfeltételezés szerint használható sorszámok apró dolgoknak tűnhetnek, de helyes megválasztásuk alapvető fontosságú a szerkesztés sima lebonyolítása szempontjából. A fenti példa kézenfekvően működik explicit sorszámok nélkül is, mivel a pillanatnyi sorszám és a sorszámok alapfeltételezés szerinti értéke mindig a kezünkre játszik. Igyekeztünk hasonló gondossággal eljárni a többi parancs esetében is.

Az **m** *(move)*, azaz áthelyezési utasítás lehetővé teszi egy vagy több sorból álló blokknak a puffer tetszőleges helyére való áthelyezését, és ily módon a „kivágás és beragasztás"-os javítást valósítja meg. A

sor1, *sor2* **m** *sor3*

parancs a *sor1*-től *sor2*-ig (ezeket is beleértve) terjedő szöveget áthelyezi a *sor3* utáni helyre. Hasonlóképpen a

.,.+1m$

parancs az aktuális, és az azt követő sort helyezi a puffer végére, a

$m0

parancs pedig az utolsó sort helyezi át a puffer elejére („a nulladik sor után"). Ha a *sor1* vagy a *sor2* sorszámot nem adjuk meg, akkor az aktuális sor helyeződik át. Az aktuális sort jelző pont az utolsó átvitt sorra mutat.

A puffer tartalmát egy tetszőleges file tartalmával egészíthetjük ki az **r** *olvasási utasítás* segítségével: az

r *file*

parancs beolvassa a *file*-t, tartalmát elhelyezi az aktuális sor után, a pontot pedig az utolsó beolvasott sorra állítja. A parancs nem változtatja meg a pufferban már elhelyezett sorokat. Ha az **r** utasításban sorszámot is megadunk, a beolvasott szöveg az ennek megfelelő sor után kerül elhelyezésre.

A puffer tetszőleges részét átírhatjuk bármelyik file-ba a **w** *írási utasítás* segítségével: a

\subroutine\, /end/w test

parancs az aktuális szubrutint a **test** nevű file-ba írja át. Ha sorokat nem jelölünk meg, a **w** parancs a puffer teljes tartalmát kiírja, ha pedig elhagyjuk a file nevét (vagyis, ha a parancs csupán egyetlen **w** betűből áll), abba a file-ba ír, melynek nevét a kezdeti

edit *file-név*

parancsban használtuk. A **w** parancs nem változtatja meg az aktuális sort és a puffer tartalmát.

Végül a szerkesztőből a **q** záróparanccsal léphetünk ki. Az a file, amelyen dolgozunk, nem őrződik meg automatikusan; ha meg akarjuk őrizni, a **q** előtt ki kell adni megfelelő **w** parancsot.

Az **a, c, i** és **q** kivételével minden szerkesztési parancs elé odaírható egy *globális előtag:*

g/*minta*/*parancs*

amely azt írja elő, hogy a parancsot a pufferben levő mindazon sorokkal végre kell hajtani, melyek tartalmazzák a *minta* egy előfordulását. A **g**, a **w**-hez hasonlóan, az alapfeltételezés szerint az összes sorra vonatkozik, de ennél kisebb intervallumot is előírhatunk. A globális előtag egyik szokásos használata mindazon sorok kinyomtatása, amelyek egy keresett mintát tartalmaznak:

g/keresett minta/p

(ez az, amit a find program is csinál), a másik pedig egy felesleges mintának a törlése minden sorból:

g/felesleges minta/d

Például a

g/% *$/d

parancs törli az üres sorokat, valamint azokat, amelyek csak szóközöket tartalmaznak. Használhatjuk például a

g/ofrmat/s//format/gp

parancsot minden **ofrmat** felkutatására, kijavítására, valamint az összes javított sor kiírására ellenőrzés céljából. Mivel a globális előtagot követő parancshoz sorszámok intervalluma is megadható, megtehetjük azt is, hogy egy minket érdeklő minta környékéről is íratunk ki sorokat:

g/érdekes minta/.—1,.+1p

A **g** előtag határozottan magasabb fokú szolgáltatás, nem ajánlható kezdő felhasználóknak, de érdemes megtanulni.

Van még egy **x** parancs is, amely olyan, mint **g** attól eltekintve, hogy azokon a sorokon működik, melyek *nem* tartalmazzák a mintát (**x** az exclude, magyarul kizárás rövidítése):

x/% *$/p

csak a nem üres sorokat nyomtatja ki.

Ezzel a parancsokat meglehetősen részletesen leírtuk, de mielőtt a programokra rátérnénk, néhány megjegyzést teszünk a sorszámokkal kapcsolatban, mivel a szerkesztés jelentős részét azoknak a soroknak a meghatározása teszi ki, amelyeken dolgozni akarunk.

A sorszámoknak a parancs többi részétől való elválasztására a vesszőhöz hasonlóan pontosvessző is használható, de ez azt is eredményezi, hogy a következő paraméter kiértékelése előtt a pont (az aktuális sor) értéke a legutolsó sorszámmal lesz egyenlő. A

/abc/;.+1p

parancs megkeresi a következő, **abc**-t tartalmazó sort, majd kinyomtatja az utána következő sorral együtt (.+1).

A sorszám-kifejezések tetszőlegesen összetettek lehetnek, de értékük 0 és $ közé kell hogy essen, a határokat is beleértve. Akárhány kifejezés állhat egymás után, csupán az a lényeges, hogy az utolsó egy vagy két kifejezés az adott parancs szempontjából megfelelő legyen. Így a

function\\ ;\\\\

parancs megkeresi a visszafelé számított második **function** deklarációt; a

/#/;//;//;//p

parancs pedig kinyomtatja a #-t tartalmazó sorok közül az aktuálistól számított harmadiktól a negyedikig terjedő valamennyi sort.

Igen sokféle szerkesztést elvégezhetünk globális előtag, pontosvesszők és többszörös szövegkörnyezet alapján történő keresések nélkül, ezért tulajdonképpen ritkán van szükségünk az utolsó példához hasonló, bonyolult szerkezetekre. De ahogy a szerkesztővel megismerkedünk, ezen tevékenységek közül egyre több és több természetessé válik. Amikor pedig előre írásban file-ok sorozatára vonatkozó komplex szerkesztési forgatókönyveket rögzítünk, ezek a lehetőségek szinte nélkülözhetetlenné válnak.

Gyakorlat

6.1 Az Olvasó hasonlítsa össze az **edit** és a saját rendszerében levő szerkesztőprogram külső tulajdonságait. □

6.2 A MEGVALÓSÍTÁS

Figyelmeztetés: Az **edit** nagy program; 900 sorával (nem számítva a **translit**-ból, **find**-ból és a **change**-ből származó részeket), 50%-kal meghaladja a könyvben szereplő bármely más program méretét. Bár mindent elkövettünk, hogy a programot jól írjuk meg és jól mutassuk be, teljes megértése mégis külön tanulmányozást igényel. Kérjük az Olvasót, hogy figyelmesen kövesse gondolatmenetünket, és ne sajnáljon a bonyolult részleteken többször is végigmenni.

A szerkesztőprogram számára adott bemenet=parancssorok sorozata, melyek mindegyike a következő alakú:

sor1, sor2 parancs akármi

ahol a *sor1, sor2* és az „*akármi*" általában elhagyható. Így a szerkesztő fő ciklusa a következő:

```
while (getlin(lin, STDIN) ¬= EOF) {
    sorszámok átvétele lin-ből
    if (státusz OK)
        a parancs végrehajtása
}
```

Már korábban megjegyeztük, hogy az **edit** azon programok közé tartozik, amelyeknek abszolút megbízhatóaknak és „elnyűhetetleneknek" *kell* lenniük. Nem fordulhat elő, hogy egy komolyabb hiba esetén egyszerűen leálljon, mivel ez a felhasználó addig befektetett munkájának elvesztését jelentheti. Olyan szerkesztőt, amely küzdelem nélkül „föladja a meccset", senki sem fog használni.

Ennek megfelelően az **edit** majdnem minden része állapot (státusz)-információt küld vissza, néha nemcsak a függvény értékeként, hanem a **status** nevű argumentumban is. Három állapotérték van: **OK** ha minden rendben van, **ERR** ha nincs minden rendben, és **EOF**, ha valamelyik alárendelt modul elfogyasztotta a teljes inputot (vagy ha az eredmény sem **OK** sem **ERR** nem volt).

A „parancs végrehajtása" egy többkimenetelű döntés, amelyben minden parancshoz külön rész tartozik. A legtöbb parancs ellenőrzi a sorszámokat, ha kell, beállítja az alapfeltételezés szerinti értékeket, majd feladatát rendszerint további szubrutinok hívásával végzi el. Mindezek azokon a rutinokon alapulnak, amelyek a pufferben levő szöveget kezelik.

Kezdjük azzal a programmal, amely a parancsok számára a sorszámokat szállítja. Ez nyilván egy különálló rész, amelyet átgondolhatunk és üzembe helyezhetünk, mielőtt bármi mást csinálnánk. Ennek segítségével, amikor elérkezik a szerkesztési tevékenységek ellenőrzésének ideje, szabadon mozoghatunk a szöveg sorai között.

A parancssort a **lin**-tömbben tároljuk, és **lin(i)** tartalmazza a következőként megvizsgálandó karaktert. A sorszámok kezelésének legfelső szintje a **getlst,** amely átveszi az input sorban szereplő sorszámokat, **i**-t úgy módosítja, hogy az az utolsó sorszám utánra mutasson, és visszaadja az állapotinformációt (**OK** vagy **ERR**) mind függvényértékként, mind pedig a **status** értékeként.

A **getlst** a sorszámok listáját a **getone** ismételt írásának segítségével olvassa, és az utolsó kettőt **line1** és **line2**-ben tárolja. Biztosítja azt is, hogy ha a sorok kijelölése nem történt meg, **line1** és **line2** egyaránt a **curln,** az aktuális sor (vagyis a pont) értékét tartalmazza. Ha egy sor volt meghatározva, akkor **line1** és **line2** egyaránt ezt tartalmazza. Emellett a **getlst** feljegyzi az aktuális sorszámok számát (0, 1 vagy 2) **nlines**-ben, és valahányszor egy pontosvessző fordul elő, módosítja **curln** értékét.

A hívások során igen sok vezérlő változó értékét kell átadni (gyakran az alárendelt modulokkal való adatközlésnek ezt a módját részesítjük előnyben), ezért az összetartozó csoportok számára kozös adatmezőket definiálunk. Amint ezt már említettük, így tudunk FORTRAN-ban legközelebb kerülni egy *struktúra* definiálásához, ami olyan összetartozó, de esetleg különböző típusú változók együttese, melyre egyetlen névvel lehet hivatkozni. Minden csoport összes deklarációját egy file-ban tároljuk, amelyet szükség szerint minden rutinba beillesztünk. A file-ok következetesen a közös adatmezőkkel azonos neveket kaptak. A sorszámokkal kapcsolatosak pedig pl. a **clines** blokkban vannak elhelyezve:

```
common /clines/ line1, line2, nlines, curln, lastln
    integer line1        # első sorszám
    integer line2        # második sorszám
    integer nlines       # a megadott sorszámok száma
    integer curln        # aktuális sor: a "pont" értéke
    integer lastln       # utolsó sor: $ értéke
```

A parancsot megelőző sorszámokat feldolgozó **getlst** programja a következő:

```
# getlst — sorszámok (ha van) átvétele lin(i)-től, i növelése
    integer function getlst(lin, i, status)
    character lin(MAXLINE)
    integer getone, min
    integer i, num, status
    include clines

    line2 = 0
    for (nlines = 0; getone(lin, i, num, status) == OK; ) {
        line1 = line2
        line2 = num
        nlines = nlines + 1
        if (lin(i) ¬= COMMA & lin(i) ¬= SEMICOL)
            break
        if (lin(i) == SEMICOL)
            curln = num
        i = i + 1
        }
    nlines = min(nlines, 2)
    if(nlines == 0)
        line2 = curln
    if (nlines <= 1)
        line1 = line2
    if (status ¬= ERR)
        status = OK
    getlst = status
    return
    end
```

A sorszámokra vonatkozó összes aritmetikai és általános helyességellenőrzés **getone**-ban történik, amely **OK**-t ad vissza egy hibátlan szám, **EER**-t hiba esetén és **EOF**-et, ha valami olyant talál, ami nem sorszám.

Miért daraboljuk szét a programot ilyen módon? Főleg ugyanazokból az okokból, mint az 5. fejezetben szereplő **match** függvényeinek elkülönítése érdekében — minden szint a vezérlés meglehetősen különböző szempontjaival foglalkozik; a szempontok egy modulban való keveredése csak zavart okozhat. Itt a **getlst** számára elég munkát ad annak számontartása, hogy hány sorszámmal találkozott, és hogy pontosvessző előfordult-e. A programnak ezen a szinten való olvasásakor igen keveset törődünk avval, hogy egy adott sorszám hogyan került átvételre, így ezt **getone**-ra hagyjuk.

Az is elég világos, hogy egy sorszám átvétele még önmagában sem egyszerű, mivel a **getone** is továbbadja a munka egy részét a neki alárendelt **getnum**-nak,

mely utóbbi egy sorszámkifejezés egyetlen tagjának összeállításával foglalkozik. Először is a **getone** következik:

```
# getone — egy sorszámkifejezés kiértékelése
  integer function getone(lin, i, num, status)
  character lin(MAXLINE)
  integer getnum
  integer i, istart, mul, num, pnum, status
  include clines

  istart = i
  num = 0
  call skipbl(lin, i)
  if (getnum(lin, i, num, status) == OK)      # első tag
      repeat {                                 # + vagy — tagok
          call skipbl(lin, i)
          if (lin(i) ¬= PLUS & lin(i) ¬= MINUS) {
              status = EOF
              break
              }
          if (lin(i) == PLUS)
              mul = +1
          else
              mul = -1
          i = i + 1
          call skipbl(lin, i)
          if (getnum(lin, i, pnum, status) == OK)
              num = num + mul * pnum
          if (status == EOF)
              status = ERR
          } until (status ¬= OK)
  if (num < 0 | num > lastln)
      status = ERR

  if (status == ERR)
      getone = ERR
  else if (i <= istart)
      getone = EOF
  else
      getone = OK

  status = getone
  return
  end
```

A **skipbl** egyszerűen kihagyja a szóközöket és a tabulátorkaraktereket; **getone**-ban arra használjuk, hogy feldolgozzuk a sorszámkifejezés tagjai között megengedett szóközöket. Használhattuk volna **ctoi**-ben (2. fejezet) és **getwrd**-ben (3. fejezet) is, mivel egy közös műveletet végez. Mostantól kezdve rendszeresen használjuk majd.

```
# skipbl — szóközök és tab-ok átlépése lin(i)-től
    subroutine skipbl(lin, i)
    character lin(ARB)
    integer i

    while (lin(i) == BLANK | lin(i) == TAB)
        i = i + 1
    return
    end
```

A **getnum** a sorszámkifejezés egy tagját értékeli ki, ahol a tag egy egész szám, . (pont), **$**, vagy szövegkörnyezet alapján való keresés.

```
# getnum — egy tag sorszámmá alakítása
    integer function getnum(lin, i, pnum, status)
    character lin(MAXLINE)
    integer ctoi, index, optpat, ptscan
    integer i, pnum, status
    include clines
    include cpat
    string digits "0123456789"

    getnum = OK
    if (index(digits, lin(i)) > 0){
        pnum = ctoi(lin, i)
        i = i - 1        # visszalépés itt, előrelépés a végén
        }
    else if (lin(i) == CURLINE)
        pnum = curln
    else if (lin(i) == LASTLINE)
        pnum = lastln
    else if (lin(i) == SCAN | lin(i) == BACKSCAN) {
        if (optpat(lin, i) == ERR)      # a minta építése
            getnum = ERR
        else if (lin(i) == SCAN)
            getnum = ptscan(FORWARD, pnum)
        else
            getnum = ptscan(BACKWARD, pnum)
        }
    else
        getnum = EOF
    if (getnum == OK)
        i = i + 1        # a következő vizsgálandó jel pozíciója
    status = getnum
    return
    end
```

A **ctoi,** melyet a 2. fejezetben ismertettünk, egy jelsorozatot alakít át egész számmá. Az **index** szintén a 2. fejezetből ismert, és megadja egy karakternek egy sorozaton belüli pozícióját vagy a 0 eredményt, ha a karakter a sorozatban nem fordul elő.

Az **optpat** felépít egy kereső mintát; ha a **lin**-ben levő minta üres, akkor az előző mintát használja. A **ptscan** hajtja végre a tényleges szövegkörnyezet alapján való keresést.

```
# optpat — a lin(i)-nél esetleg található minta felépítése
integer function optpat(lin, i)
character lin(MAXLINE)
integer makpat
integer i
include cpat

if (lin(i) == EOS)
        i = ERR
else if (lin(i + 1) == EOS)
        i = (ERR
else if lin(i + 1) == lin(i))          # ismételt elhatároló
        i = i + 1                      # a már meglevő minta megmarad
else
        i = makpat(lin, i + 1, lin(i), pat)
if (pat(1)== EOS)
        i=ERR
if (i==ERR) {
        pat(1)=EOS
        optpat=ERR
        }
else
        optpat=OK
return
end
```

Az **else—if** lánc biztosítja, hogy a vizsgálatok pontosan a megfelelő sorrendben kerülnek végrehajtásra (pl. nem kívánjuk **lin(i+1)**-et megnézni, ha **lin(i) EOS**). Ebben a tekintetben a vizsgálati lánc jobb, mint a **case** utasítás (amelyet néhány nyelv használ), mivel a vizsgálatok sokkal általánosabbak lehetnek, és a kiértékelés sorrendje is irányított.

A **makpat**-ot és az őt támogató rutinokat az 5. fejezetben tárgyaltuk. Annak idején a **makpat**-ot úgy írtuk meg, hogy a vizsgálat leállításához tetszőleges elhatárolót használhasson. Az **optpat** az első alkalom, ahol ezt a lehetőséget kihasználjuk. Itt az elhatároló a **lin(i)**-ben levő karakter, ami szövegkörnyezet alapján való keresésnél vagy "/" vagy "\".

A **ptscan** a haladási iránytól függő .+1-től vagy .−1-től kezdve körbehaladva végigvizsgálja a puffert, amíg vagy egy illeszkedést nem talál, vagy vissza nem ér a **curln**-hez. A keresés az aktuális sor utáni vagy előtti sornál kezdődik, mivel feltehetően éppen előzőleg csináltunk valamit az aktuális sorral, és a következő sorral akarjuk folytatni a munkát. Az illeszkedő minta keresését a **match** és alárendeltjei végzik, melyeket szintén az 5. fejezetben készítettünk el.

```
# ptscan — a minta következő előfordulásának keresése
    integer function ptscan(way, num)
    integer gettxt, match, nextln, prevln
    integer k, num, way
    include clines
    include cpat
    include ctxt

    num = curln
    repeat {
        if (way == FORWARD)
            num = nextln(num)
        else
            num = prevln(num)
        k = gettxt(num)
        if (match(txt, pat) == YES) {
            ptscan = OK
            return
            }
        } until (num == curln)
    ptscan = ERR
    return
    end
```

Az **optpat**-nak és a **ptscan**-nak ismernie kell a **cpat** nevű közös adatmezőben szereplő **pat** mintatömböt:

```
common /cpat/ pat(MAXPAT)
    character pat    # minta
```

Ezen túlmenően, a **ptscan** elő kell hogy állítsa a **match** számára a tényleges szövegsorokat. Ez a **gettxt** hívásának segítségével végzi el feladatát, amely a szövegsort a **ctxt** közös adatmezőben szolgáltatja:

```
common /ctxt/ txt(MAXLINE)
    character txt    # az illesztésre és kivitelre kerülő sor
```

A **ptscan** a sorszámot indexszé alakítja, melyet a sorokat átrendező rutinok használnak. Mivel ezt az információt a **ptscan** nem használja, megvalósítását ezen a szinten később tárgyaljuk.

A **prevln** és a **nextln** függvények soronként járják körbe a puffert. Kitűnt, hogy célszerű a pufferben állandó jelleggel egy „nulladik sort” tartani, amely nem tartalmaz semmit és így semmi sem illeszkedik hozzá, és amely szabályos sorszámmal rendelkező sor lehet az **a, m** és **r** parancsok számára, melyeknek az első sor elé kell tudniuk valamit elhelyezni. A nulladik sor példa egy hasznos technikai megoldásra, amely úgy egyszerűsíti a programot, hogy mesterséges elemet kapcsol az adatstruktúrához, a határesetek kezelésének egyszerűsítése céljából.

```
# nextln — a "line" utáni sor elérése
    integer function nextln(line)
    integer line
    include clines

    nextln = line + 1
    if (nextln > lastln)
        nextln = 0
    return
    end

# prevln — a "line" előtti sor elérése
    integer function prevln(line)
    integer line
    include clines

    prevln = line - 1
    if (prevln < 0)
        prevln = lastln
    return
    end
```

A sorszámmal foglalkozó kód összefoglalásaképpen bemutatjuk az eddigi fontosabb szubrutinok és függvények hívási fáját.

```
edit
    getlst
        getone
            getnum
                optpat
                    makpat
                ptscan
                    gettxt, match, nextlin, prevln
```

Most is az általánostól (**getlst**) a specifikus (**optpat, ptscan**) felé haladunk, több lépésben. A hierarchia minden szintjén a teljes problémának egy fokozatosan csökkenő méretű részét kezeljük, kiküszöbölve ily módon azt, hogy az egyes szinteken túl sok részletet kelljen ismerni.

A **makpat** és a **match** további rutinokat hív, melyeket a 2. és az 5. fejezetekben írtunk meg. Ez természetesen számottevő programozási munkát takarít meg a számunkra, de még ennél is fontosabb az egységes szemlélet. A **translit, find, change** és **edit** ugyanazokat a szabályokat és konvenciókat használják a mintákkal kapcsolatban; nincs szükség arra, hogy mindegyikükhöz külön szabályokat tanuljunk és tartsunk fejben. Ez egyszerűsíti a felhasználók feladatát, és azokat, akik egy programot már ismernek, a többi kipróbálására ösztönzi.

Gyakorlat

6.2 Írjunk egy főprogramot, amely beolvassa a parancssorokat és meghívja a **getlst**-et. Írjunk egy ál **gettxt** rutint és definiáljuk a **buf**-ot úgy, hogy két vagy három előre meghatározott szövegsorhoz hozzá tudjunk férni. Vizsgáljuk meg a **getlst**-et és alárendeltjeit ebben a minimális tesztelési környezetben! □

6.3 A VEZÉRLŐPROGRAM

Térjünk most vissza a legfelső szintre, és tárgyaljuk meg a parancskezelés részleteit. Alapjában véve a szerkesztő egy ciklus, amely parancssorokat olvas, dekódolja azokat, majd végrehajtja őket. Amennyire csak lehet, biztosítani kívánjuk, hogy minden parancs kritikusan vizsgálja meg a számára előírt tevékenységet, nehogy egy vissza nem fordítható művelettel teljes szövegfile-okat tehessen tönkre. Ez azt jelenti, hogy akárcsak a 2. fejezetbeli **expand** esetében, az **edit** vezérlési szerkezete is főleg hibaellenőrzést tartalmaz.

Az **edit** soronként csak egyetlen parancsot fogad el, bár ezt esetleg egy globális előtag is megelőzheti. A globális feldolgozás részleteit később tárgyaljuk, miután részletesebben foglalkoztunk az alapparancsokkal. Pillanatnyilag az **edit** fő feldolgozási ciklusát a következőképpen írhatjuk meg:

```
while (getlin(lin, STDIN) ¬= EOF) {
     i = 1
     cursav = curln
     if (getlst(lin, i, status) == OK) {
          if (ckglob(lin, i, status) == OK)
               status = doglob(lin, i, cursav, status)
          else if (status ¬= ERR)
               status = docmd(lin, i, NO, status)
          # különben hiba, nincs mit tenni
          }
     if (status == ERR) }
          call remark("?")
          curln = cursav
          }
     else if (status == EOF)
          break
     # különben OK, a ciklus folytatódhat
     }
stop
```

A **ckglob** a **g/.../** vagy az **x/.../** parancsokat keresi; ha valamelyiket megtalálja, a sorokat a **doglob**-bal való feldolgozás céljából megjelöli, amely a megjelölt sorokon végrehajtja a kívánt parancsokat; ezekhez később visszatérünk. Pillanatnyilag tegyük fel, hogy van egy üres **ckglob**, amely mindig **EOF**-et ad vissza (jelezve, hogy nem talált globális utasítást). Ha nincs globális előtag és nem fordult elő hiba sem, a **docmd** végrehajtja a parancsot a **getlst** által meghatározott sor-intervallumra. A **docmd NO** paramétere azt jelenti, hogy globális előtaggal nem rendelkező parancs végrehajtása során került hívásra.

A főprogram, hiba esetén vissza kell hogy állítsa **curln**-t, mivel az a **getlst** által talált minden pontosvesszőnél megváltozik, valamint a **doglob** és a **docmd** által végrehajtott parancsok is módosíthatják.

Mivel a szerkesztőt párbeszédes üzemmódra terveztük, az minden hibára egyetlen kérdőjellel reagál. A tömörség helyénvaló, mivel a hiba majdnem mindig nyilvánvaló; rendszerint egy elütés a gépelésben vagy egy sikertelen keresés. Ilyen esetekben a hibaüzenet lassítaná a munkát. Azonban az **edit** olyan szerkezetű, hogy részletesebb

hibaüzenetek könnyen beleilleszthetők; az egyik gyakorlat éppen ennek részletesebb kidolgozásával foglalkozik.

A **docmd** legnagyobb része egy hosszú **if...else if...else if** szerkezet, amely a végrehajtandó parancsot azonosítja. Minden ág a feladat elvégzését leíró néhány sorból áll, ami leggyakrabban egy szubrutinhívás.

A *kiíró* **(p)** utasításhoz tartozó programrész például a következő:

```
else if (lin(i) == PRINT) {
    if (lin(i + 1) == NEWLINE)
      andif (defalt(curln, curln, status) == OK)
          status = doprnt(line1, line2)
    }
```

Ez ellenőrzi, hogy a parancs formailag hibátlan-e, meggyőződik arról, hogy a sorszámok ésszerűek-e, majd végrehajtja a megfelelő rutint. Az **andif** az **if** szinonimája; használatával azt a kívánságunkat akartuk kifejezésre juttatni, hogy a vizsgálatokat az adott sorrendben kell végrehajtani. Amint ezt a 2. fejezetben tárgyaltuk, a FORTRAN, sok más nyelvhez hasonlóan, nem biztosít semmilyen rögzített sorrendet logikai kifejezések kiértékelésekor (más kifejezések kiértékelésekor sem), és nem biztosítja azt sem, hogy a logikai kifejezés kiértékelése befejeződjék, amint annak igazságértéke ismertté válik. Így például az

```
if (lin(i + 1) == NEWLINE & defalt(curln, curln, status) == OK)
```

feltételt nem használhattuk, mivel nem akartuk, hogy a **defalt** meghívódjék és a **status**-t OK-ra állítsa, miután bebizonyosodott, hogy **lin(i+1)** nem **NEWLINE**. Az **andif** után soha nem következik a szimmetriát visszaállító **else**, ezért óvatosságból mindig kapcsos zárójelbe kell tennünk az **if...andif** szerkezetet.

A **p** parancshoz két sorszám tartozik. Ha csak egy van, ezt használjuk mindkét sorszám helyén (azaz csak egyetlen sort nyomtatunk ki). Ha egy sincs, az aktuális sort használjuk mindkét sorszám helyén. Erre használt jelölésünk **(.,.)p**, ahol a zárójelek azt mutatják, hogy a sorszámok elhagyhatók, a két pont pedig az alapfeltételezést fejezi ki. A **defalt** beállítja az alapfeltételezés szerinti sorszámokat:

```
# defalt — sorszámok alapfeltételezésének beállítása
    integer function defalt(def1, def2, status)
    integer def1, def2, status
    include clines

    if (nlines == 0) {
        line1 = def1
        line2 = def2
        }
    if (line1 > line2 | line1 <= 0)
        status = ERR
    else
        status = OK
    defalt = status
    return
    end
```

A nulladik sor kinyomtatása vagy a puffer végének túllépése semmilyen körülmények között nem megengedett, ezért ilyenkor a **defalt** hibát regisztrál. Figyeljük meg azt is, hogy **defalt** megvizsgálja, nem teljesül-e a **line1<=0** feltétel annak ellenére, hogy **getone** biztosítja, hogy **line1** nem lehet negatív. Ez megint csak defenzív programozás — a program valamely más részéről jövő hiba ellen véd.

A tényleges nyomtatás már egyszerű:

```
# doprnt — "from"-tól "to"-ig terjedő sorok nyomtatása
  integer function doprnt(from, to)
  integer gettxt
  integer from, i, j, to
  include clines
  include ctxt

  if (from <= 0)
      doprnt = ERR
  else {
      for (i = from; i <= to; i = i + 1) {
          j = gettxt(i)
          call putlin(txt, STDOUT)
          }
      curln = to
      doprnt = OK
      }
  return
  end
```

Mivel a **doprnt** különböző helyekről kerül hívásra a **docmd**-n belül, a **defalt**-hoz hasonlóan itt is ellenőrizni kell a nulladik sort. A tényleges szöveg elérésére ugyanazt a mechanizmust használjuk, mint a **ptscan**-ben, a kinyomtatásra kerülő sort **gettxt** hívásával állítjuk elő. A **gettxt** által visszaadott indexre most sincs szükségünk.

Figyeljük meg, hogy az utolsó sor kiírása után nem kerül sor egy felesleges „file vége"-szerű üzenet kinyomtatására. Az **edit** sok esetben tényleg szűkszavú. A bőbeszédű programok egyik baja az, hogy nem lehet őket elhallgattatni, amikor más programmal akarjuk együtt használni. Csak akkor nyomtat, ha ezt előírjuk, és ezért a parancsok üzenetek nélkül is tudnak működni. Azonban a nyomtatást elég gyakran igénylik ahhoz, hogy érdemes legyen valamilyen külön jelölést és rövidítést bevezetni a számára. Az **s, m** és **d** parancsok után állhat egy **p** parancs, amely az utolsó érintett sor kinyomtatását írja elő. Továbbá, a csupán sorszámokból álló parancssor (amelyben nem fordul elő parancs) az utolsó sorszámmal specifikált sor kinyomtatását eredményezi. Ez leggyakrabban egyetlen sorszámból áll, mint pl. **1** vagy **$** vagy **/abc/—2,** de tartalmazhat több sorszámot is:

```
/abc/;//;//;//
```

amelynek hatására az **abc**-nek csupán a negyedik előfordulását tartalmazó sor íródik ki, nem pedig a harmadik és a negyedik, mint a sorvégi **p** használata esetén. Végül pedig egy teljesen üres parancssor (ami csak az újsor-karakterből áll) **.+1p**-ként értelmeződik, így újsor-karakterek leütésével végig tudunk menni a pufferen.

A **docmd** az első program a könyvben, amely hosszabb, mint egy oldal. Bár önkényes határok felállítása nem ésszerű, mégis célszerűnek tűnik, hogy az egyes

rutinok mérete egy oldalnál kisebb legyen, mivel annál könnyebb egy programot megérteni, minél rövidebb. (És mihelyt az egyoldalas határt átléptük, nehézkessé válik a bekezdési mélységek nyomon követése.) Rutinjaink hosszának mediánja 20 sor; átlaga 15 sor. Még az egy lapnál nagyobb szubrutinjaink tervezésekor is gondosan ügyeltünk arra, hogy azok könnyen érthetők legyenek, mindegyik csupán egy **else if** láncból áll, amelyek nagyszámú lehetőség közül választanak, és minden egyes választás könnyen áttekinthető. A teljes **docmd** programot a fejezet későbbi részében adjuk, itt a nyomtatás vezérlési részét mutatjuk be:

```
# docmd — parancsok feldolgozása, globálist kivéve (nem teljes)
  integer function docmd(lin, i, glob, status)

  pflag = NO              # d, m vagy s állíthatja
  status = ERR
  if (lin(i) == APPENDCOM) {
      hajtsd végre az append parancsot
      }
  # és így tovább a többi parancsra
  else if (lin(i) == PRINT) {
      if (lin(i + 1) == NEWLINE)
        andif (defalt(curln, curln, status) == OK)
            status = doprnt(line1, line2)
      }
  else if (lin(i) == NEWLINE) {
      if (nlines == 0)
            line2 = nextln(curln)
      status = doprnt(line2, line2)
      }
  else if (lin(i) == QUIT) {
      if (lin(i + 1) == NEWLINE & nlines == 0 & glob == NO)
            status = EOF
      }
  # különben status = ERR
  if (status == OK & pflag == YES)
      status = doprnt(curln, curln)
  docmd = status
  return
  end
```

A programrész tartalmazza a **q** kilépő utasítás programját is, amely az egyetlen **q** betűt tartalmazó parancssor hatására kerül hívásra. Ennek eredményeként a szerkesztőből kilépünk, éppúgy, mintha a parancsok olvasása során **EOF**-fel találkoztunk volna. Ugyanabból az okból, amiért a legtöbb parancs néma, most sem nyomtatódik ki semmi.

Továbbá, a **q** parancs után egyik file-ba sem íródik semmi. Felvethető, hogy jobb lenne a szerkesztő puffert automatikusan kiírni, vagy legalábbis a felhasználó beleegyezését kérni kilépés előtt. Nehéz eldönteni, hogy mennyire szükséges a felhasználókat saját balfogásaiktól megvédeni, de tapasztalatunk szerint általában a legokosabb nem keresztezni mások útjait: fel kell tennünk, hogy tudják, mit csinálnak, és hagynunk kell őket a lehető legkevesebb korlátozással és figyelmeztetéssel dolgozni.

Például az **edit** egyik gyakori felhasználása egy file-nak a pufferban való megvizsgálása és esetleges módosítása a visszaírás minden szándéka nélkül. Az a szerkesztő, amely mindenképpen elvégzi a visszaírást, rosszul tervezett, mivel jelentős mértékben korlátozza az ilyenfajta szerkesztést. Még a döntésre való felszólítás is zavaró lehet, ellentmondásba kerülhet egy előre rögzített szerkesztési forgatókönyvvel.

Ellenvetheti valaki, hogy a 3. fejezetben szereplő **archive** program megtagadta az archívum összes file-jának törlését, amikor ezt implicit módon írták elő. Következetlenek vagyunk tehát? A különbség az, hogy az **archive** egy meglehetősen speciális, értékes információ hosszú idejű tárolására szolgáló eszköz, amit az átlagos felhasználó valószínűleg nem vesz kezébe minden nap. Ezért az **archive** minden cselekedetében óvatos. Ezzel szemben az **edit** a rendszert a külvilággal összekapcsoló, állandóan használt, általános célú eszköz. Ez nyugodtabban veheti a felhasználó parancsait készpénznek.

6.4 PUFFER-REPREZENTÁCIÓ

A **docmd**-ben körvonalazott másik parancs a hozzáillesztő utasítás, amely szöveget helyez el a pufferban. Az elhelyezendő sorok a pufferban közvetlenül a megadott sor mögé, vagy — ha sorszám nincs megadva — közvetlenül az aktuális sor mögé kerülnek. Ezt az elvet a **(.)a** rövidítéssel fejezzük ki. Világosan kell látni, hogy miért határozzuk meg ilyen aprólékosan az elfogadható parancs fogalmát. Ha a felhasználó elfelejt a szöveg leírása előtt belépni a hozzáillesztési üzemmódba, nyilván nem akarja, hogy egy szó betűi véletlenszerűen megváltoztassák a szöveget. Egy kérdőjel hamarosan figyelmezteti az első értelmezhetetlen sor után.

A **docmd** azon része, amely **append**-et hívja, a következő:

```
if (lin(i) == APPENDCOM) {
     if (lin(i + 1) == NEWLINE)
          status = append(line2, global)
     }
```

maga az **append** pedig

```
# append — sorok csatlakoztatása "line" után
     integer function append(line, glob)
     character lin(MAXLINE)
     integer getlin, inject)
     integer line, glob
     include clines

     if (glob == YES)
          append = ERR
     else {
          curln = line
```

```
    for (append = NOSTATUS; append == NOSTATUS; )
        if (getlin(lin, STDIN) == EOF)
            append = EOF
        else if (lin(1) == PERIOD & lin(2) == NEWLINE)
            append = OK
        else if (inject(lin) == ERR)
            append = ERR
    }
return
end
```

A **NOSTATUS** értéke nem lehet egyenlő **OK**-val, **ERR**-rel vagy **EOF**-fel. A globális előtag figyelembevételével végrehajtott szövegbeillesztés komplikációkat okozhat, ezért most nem tárgyaljuk. Tehetjük ezt annál is inkább, mivel egy **r** paranccsal ugyanaz a hatás elérhető. Az **append** jelen változata nem engedi meg a globális beillesztést, de nincs kizárva annak a lehetősége, hogy a későbbiekben ennek megvalósítására is sor kerülhessen.

A tényleges munkát, melyet az egyes új sorok beillesztése (és a **curln** megfelelő módosítása jelent), az **inject** rutin végzi. Működésének megértéséhez többet kell tudnunk a puffer szervezéséről.

Azt akarjuk elérni, hogy sorokat szabadon átrendezhessünk, a keresést pedig mindkét irányban haladva hatékonyan végezhessük. Alkalmazhatnánk a szövegsorokra hivatkozó egyetlen mutatótömböt és ezt rendeznénk át a sorok törlésekor vagy áthelyezésekor, ahogyan ezt a 4. fejezetbeli **sort** esetében tettük. Mivel azonban a beillesztés, a törlés és az olvasás egyaránt megváltoztatja a sorok számát, a mutatóknak egy kompakt halmazba való szervezése némi külön adminisztrációt igényel. Ezért a következőkben megvizsgáljuk a szövegsorok kétszeresen kapcsolt listáját, melyben minden tag (azaz egy szövegsor) az előző és a következő sorra mutató pointereket tartalmaz. (Azért használunk kétszeresen kapcsolt listát, hogy az olyan műveletek, mint pl. a hátrafelé keresés, ne legyenek időigényesebbek.) Ilyen módon a sorok szükség szerinti áthelyezése a mutatók tartalmának módosításával oldható meg. Ugyancsak könnyen elvégezhető teljes részek beillesztése és törlése is.

Hol tároljuk a sorokat? Legegyszerűbb volna mindent az operatív tárban tartani, de ez komoly mértékben korlátozná a kezelhető file-ok méretét. Kompromisszumos megoldás lehet az, hogy a szöveget egy külső munkafile-ban tároljuk, az operatív tárban pedig csak a listákat és azt az információt tartjuk, ami a szövegsoroknak a file-ban való eléréséhez szükséges. Ezzel korlátoztuk az összes kezelhető sorok számát, de sokkal több karakterből álló szöveg feldolgozására vagyunk képesek. A másik lehetőség, hogy a teljes kapcsolt listát egy file-ban tartjuk, melynek az éppen szükséges részéhez egy rögzített méretű „ablakon” keresztül férünk hozzá.

Bármelyik megoldást választjuk is, fontos, hogy azt, amennyire csak lehet, elszigeteljük a program egészétől. Ily módon lehetőség van egy adott megvalósítás optimalizálására, vagy akár az egész hozzáférési stratégia megváltoztatására néhány alacsony szintű pufferkezelő rutin módosítása árán.

Jelen esetben a fenti lehetőségek bármelyikét megvalósíthatjuk 6 függvény megírásával:

setbuf: beállítja a puffer kezdőértékét, hogy az csak egyetlen nulladik sort tartalmazzon és megnyit egy munkafile-t, ha szükséges.

clrbuf: megszünteti a munkafile-t, ha használtunk ilyent.

inject(lin): a **lin**-ben levő szöveget a pufferba másolja közvetlenül az aktuális sor után, új sort kezd ott, ahol az újsor-karakterek szerint ez szükséges és **curln**-t az utolsó beillesztett sorra állítja.

getind(n): az n sorszámhoz egy egyedi indexet rendel, amelynek alapján a kérdéses sorra vonatkozó összes információ elérhető.

gettxt(n): elvégzi ugyanazt a leképezést, mint **getind**, de egyben a sor tartalmát át is másolja a **ctxt** közös adatmezőbeli **txt** tömbbe.

relink(k1, k2, k3, k4): a listát olyan módon rendezi át, hogy **k2** visszamutasson **k1**-re, **k3** pedig előremutasson **k4**-re, ahol a **ki**-k a **getind** vagy **gettxt** hívásokból származó indexeket jelölnek.

A **k** indexekre a sorszámok mellett azért van szükség, hogy nyilvántarthassuk a sorokat az átrendezések folyamán, amikor a sorszámok jelentése többször is megváltozhat.

Nem kell (és nem is áll módunkban) tudni, hogy az egész szöveg, a listák vagy bármi az operatív tárban van-e. Mindaddig, amíg a szöveg egy sora és a **getind**, ill. **gettxt** által szolgáltatott utolsó négy listaelem rendelkezésre áll, minden szerkesztési tevékenységet el tudunk végezni. Ez azt jelenti, hogy az operatív tárban levő puffer tartalmazhatja az összes adatot vagy lehet egy sokkal „nagyobb világra" nyíló, rögzített méretű „ablak". A lényeg az, hogy ha már van egy működő szövegszerkesztőnk, sok mindent el tudunk intézni, mivel a pufferkezelővel való kapcsolat jól körülhatárolt, a hozzá tartozó program pedig elkülönített.

Hangsúlyozzuk, hogy nem ebből a hat rutinból indultunk ki. A szerkesztő felülről lefelé való megírása folyamán „fedeztük fel" őket. Minden egyes lépésnél látva, hogy a szövegen milyen műveleteket kívánunk végrehajtani, kialakítottuk a fenti néhány alapfüggvényt. Az is segítségünkre volt, hogy a tényleges megvalósításra vonatkozó döntést, amennyire csak lehetett, késleltetni akartuk. Ez a célkitűzés jónéhány, a jelenleginél több korlátozást magában foglaló programszerkezettől óvott meg minket. Ezen alapművelet-készlet és olyan mérések birtokában, melyek megmutatják, hol tölti idejét a szerkesztőprogram, az **edit** hatásfokát, ha ez szükséges, oly módon javíthatjuk, hogy a program nagy részéhez nem nyúlunk hozzá.

Az első cél azonban egy működő szerkesztőprogram létrehozása. Ennek érdekében kiválasztjuk a legkönnyebben megvalósítható pufferkezelést: mindent az operatív tárban tartunk a **buf** tömbben. A lista egy elemének szerkezete a következő:

```
buf(k + 0)    PREV     az előző sor indexe
buf(k + 1)    NEXT     a következő sor indexe
buf(k + 2)    MARK     a globális módosító használja
buf(k + 3)    TEXT     a szöveg első karaktere
buf(k + 4)    ...      második karakter stb.
```

Minden szövegdarab egy **EOS**-sel fejeződik be. A **cbuf** közös adatmező a következőket tartalmazza:

```
common /cbuf/ buf(MAXBUF), lastbf
    character buf        # mutató és szövegpuffer
    integer lastbf       # a buf-ban használt utolsó elem
```

A szövegsorok szükség szerint másolódnak át a **txt** pufferba, amely a **ctxt** közös adatmezőben található. A **ctxt**-t a **ptscan**-nel kapcsolatban mutattuk be, de felhasználtuk a **doprnt**-nél is. A teljesség kedvéért újra itt látható:

```
common /ctxt/ txt(MAXLINE)
    character txt          # az illesztésre és kivitelre váró sor
```

Ezek után megírhatjuk a pufferkezelő rutinoknak az operatív táron működő változatait. A **clrbuf** a legkönnyebb, amikor nincs munkafile:

```
# clrbuf (operatív tárbeli) kezdőérték új file esetén
    subroutine clrbuf

    return        # tennivaló nincs
    end
```

A **getind** és a **gettxt** a megadott sort a lista elejétől való bejárásával találják meg. A **getind** csak az indexet találja meg; a **gettxt** a **getind**-et használja a sor megkeresésére, majd visszaadja a szöveget is. Ezek azért külön függvények, mivel sok szerkesztőparancs a sorok tartalmától függetlenül, a sorok indexein való műveletekkel végrehajtható. Amikor később a szöveg tárolására egy munkafile-t használunk, ez az elkülönítés fogja lehetővé tenni, hogy csak akkor forduljunk a file-hoz, ha ez tényleg szükséges.

```
# getind — az index megkeresése a pufferben
    integer function getind(line)
    integer j, k, line
    include cbuf

    k = LINE0
    for (j = 0; j < line; j = j + 1)
        k = buf(k + NEXT)
    getind = k
    return
    end
```

Ebben a megvalósításban **LINE0** természetesen 1.

```
# gettxt (operatív tárbeli) "line"-hez tartozó szöveg elérése és átvétele
    integer function gettxt(line)
    integer getind
    integer line
    include cbuf
    include ctxt

    gettxt = getind(line)
    call scopy(buf, gettxt + TEXT, txt, 1)
    return
    end
```

A **gettxt** a 3. fejezetben elkészített **scopy**-t használja a sornak **txt**-be való bemásolására.

A **relink** csak két kapcsolást módosít; sok feladatban többször kell hívni, de vannak helyzetek, amikor erre a korlátozott tevékenységre van szükség.

```
# relink — két kapcsolás módosítása
    subroutine relink(a, x, y, b)
    integer a, b, x, y
    include cbuf

    buf(x + PREV) = a
    buf(y + NEXT) = b
    return
    end
```

A **setbuf** a puffer kezdeti beállítását végzi el oly módon, hogy létrehozza a nulladik sort és azt önmagához kapcsolja. A nulladik sor csupán egy **EOS**-ből áll, így ehhez semmi nem illeszkedik, de a jelenléte szabályossá teszi a programot és így például a nulladik sor után való beillesztés nem speciális eset.

```
# setbuf (operatív tárbeli) — a sortároló puffer kezdő állapotának
                                # beállítása
    subroutine setbuf
    integer addset
    integer junk
    include cbuf
    include clines

    call relink(LINE0, LINE0, LINE0, LINE0)
    lastbf = LINE0 + TEXT
    junk = addset(EOS, buf, lastbf, MAXBUF)
    curln = 0
    lastln = 0
    return
    end
```

Az **inject** főleg azért bonyolult, mert olyan sorokat tud kezelni, melyek belső újsor-karaktereket tartalmaznak. Ezt még nem említettük, de ha visszagondolunk az 5. fejezetben levő **change** programra, belátjuk, nincs okunk megtiltani, hogy a sor közepén valamit egy újsor-karakterre változtassunk. Ha meg akarjuk őrizni ezt a hasznos tulajdonságot az **s** parancsban, az **inject**-nek szét kell darabolnia a „sor"-t belső újsor-karakterekkel, a kapott külön sorokat pedig egyenként kell beilleszteni a pufferba. Az **inject** emellett elvégzi a **curln** és a **lastln** megfelelő módosítását.

```
# inject (operatív tárbeli) — szövegbeillesztés curln után
    integer function inject(lin)
    character lin(MAXLINE)
    integer addset, getind, nextln
    integer i, junk, k1, k2, k3
    include cbuf
    include clines
```

```
    for (i = 1; lin(i) ¬= EOS;) {
        k3 = lastbf
        lastbf = lastbf + TEXT
        while (lin(i) ¬= EOS) {
            junk = addset(lin(i), buf, lastbf, MAXBUF)
            i = i + 1
            if (lin(i - 1) == NEWLINE)
                break
            }
        if (addset(EOS, buf, lastbf, MAXBUF) == NO) {
            inject = ERR
            break
            }
        k1 = getind(curln)
        k2 = getind(nextln(curln))
        call relink(k1, k3, k3, k2)
        call relink(k3, k2, k1, k3)
        curln = curln + 1
        lastln = lastln + 1
        inject = OK
        }
    return
    end
```

Most a szubrutin-hierarchia további részének bemutatásával megkíséreljük megkönnyíteni az Olvasó számára a hívások közti eligazodást.

```
edit
    setbuf
        relink
    getlst
    docmd
        append
            getlin
            inject
                getind, nextln, relink
        defalt
        doprnt
            gettxt, putlin
    clrbuf
```

A további bővítések legnagyobb része a **docmd**-t érinti.

Gyakorlatok

6.3 Valósítsuk meg a szövegszerkesztőt ezekkel a pufferkezelő rutinokkal, és az **edit** eddig bemutatott részével, majd próbáljuk ki! A **ckglob** és a **doglob** helyett üres rutinokra lesz szükségünk. Mit kell ezeknek végrehajtaniok? Működniük kell a szövegbeillesztésnek, a különböző sorok kinyomtatásának, a szövegkörnyezet alapján való keresésnek és a kilépésnek. Győződjünk meg róla, hogy a program minden zugába betekintettünk! □

6.4 Hogyan vinnénk be egy egyetlen pontból álló sort a pufferba? □

6.5 Mi történik az aktuális sorral, ha „semmit" illesztünk be, azaz a következő parancsot írjuk:

```
a
.
```

Ésszerű-e, hogy ez történik? □

6.6 Előállhat-e olyan helyzet, amelyben a szövegkörnyezet alapján való keresés a nulladik sorhoz illeszkedik? □

6.5 TOVÁBBI PARANCSOK: TÖRLÉS, BESZÚRÁS, CSERE, SORSZÁMNYOMTATÁS, ÁTVITEL

A beillesztés mellett egy másik fontos funkció a nem kívánt szöveg törlése. Ezt a törlési parancs, a **(.,.)d** végzi. A **docmd**-t a következővel egészítjük ki:

```
else if (lin(i) == DELCOM) {
      if (ckp(lin, i + 1, pflag, status) == OK)
        andif (defalt(curln, curln, status) == OK)
        andif (delete(line1, line2, status) == OK)
        andif (nextln(curln) ¬= 0)
              curln = nextln(curln)
        }
```

A törlési parancs eltávolítja a megadott sort vagy sorokat, a **curln**-t pedig az eltávolított sorok utáni első sorra állítja, hacsak ez nem esne kívül a pufferen, mely esetben a **curln** felveszi a **lastln** értékét. Ezután vizsgálat következik, hogy nem áll-e az utasítás végén egy esetleges **p**, amely ennek a sornak ellenőrzési célból történő kinyomtatását írja elő. Ha van, a **ckp** rutin jelzést ad **pflag**-ben:

```
# ckp — parancs utáni "p" vizsgálata
      integer function ckp(lin, i, pflag, status)
      character lin(MAXLINE)
      integer i, j, pflag, status

      j = i
      if (lin(j) == PRINT) {
            j = j + 1
            pflag = YES
            }
```

```
else
    pflag = NO
if (lin(j) == NEWLINE)
    status = OK
else
    status = ERR
ckp = status
return
end
```

A tényleges munkát a **delete** függvény végzi el, amely **curln**-t az eltávolított sorok elé állítja, és visszaállítja **lastln**-t az új utolsó sorra.

```
# delete — sorok törlése "from"-tól "to"-ig
  integer function delete(from, to, status)
  integer getind, nextln, prevln
  integer from, k1, k2, status, to
  include clines

  if (from <= 0)
      status = ERR
  else {
      k1 = getind(prevln(from))
      k2 = getind(nextln(to))
      lastln = lastln — (to — from + 1)
      curln = prevln(from)
      call relink(k1, k2, k1, k2)
      status = OK
      }
  delete = status
  return
  end
```

A másik három parancs könnyen megvalósítható a már meglevő rutinok segítségével. A beszúrás, azaz az **(.)i** parancs a szöveget közvetlenül az adott sorszámú sor elé helyezi el.

```
else if (lin(i) ==INSERT) {
    if (lin(i + 1) == NEWLINE)
        status = append(prevln(line2), glob)
    }
```

A csere, azaz a **(.,.)c** parancs törli az adott intervallumba eső sorokat, majd helyükre szöveget tesz.

```
else if (lin(i) == CHANGE) {
    if (lin(i + 1) == NEWLINE)
      andif (defalt(curln, curln, status) == OK)
      andif (delete(line1, line2, status) ==OK)
          status = append(prevln(line1), glob)
    }
```

A **(.)=** parancs egy sorszámkifejezés értékének (vagy az aktuális sorszámnak) kinyomtatását eredményezi, így megnézhetjük, hogy valamelyik sor hol helyezkedik el a szövegben. Leggyakrabban **$=** formában használják, annak kiírására, hogy hány sor van a pufferban.

```
else if (lin(i) == PRINTCUR) {
     if (ckp(lin, i + 1, pflag, status) == OK) {
          call putdec(line2, 1)
          call putc(NEWLINE)
          }
     }
```

Az **m** áthelyezési parancs szövegsorokat rendez át: a

(.)m *sor3*

parancs a megadott sort vagy sorokat kiemeli pillanatnyi helyükről és közvetlenül *sor3* után helyezi el. Mivel a *sor3* átvételét a **getone** intézi, bármilyen helyes kifejezést alkalmazhatunk, pl. a

```
/format/m/end/−1p
```

parancs a következő, **format**-ot tartalmazó sort viszi át közvetlenül a következő **end**-et tartalmazó sor elé. A **curln** változatlanul az utoljára átvitt sorra mutat, ez esetben arra, amely a **format**-ot tartalmazza. Ezt a sort az elhagyható sorvégi **p** kinyomtatja. A **docmd**-ben levő megfelelő programrész a következő:

```
else if (lin(i) == MOVECOM) {
     i = i + 1
     if (getone(lin, i, line3, status) == EOF)
          status = ERR
     if (status == OK)
       andif (ckp(lin, i, pflag, status) == OK)
       andif (defalt(curln, curln, status) == OK)
          status = move(line3)
     }
```

A munkát pedig a **move** végzi el:

```
# move — line1-től line2-ig terjedő szöveg átvitele line3 utánra
     integer function move(line3)
     integer getind, nextln, prevln
     integer k0, k1, k2, k3, k4, k5, line3
     include clines

     if (line1 <= 0 | (line1 <= line3 & line3 <= line2))
          move = ERR
     else {
          k0 = getind(prevln(line1))
          k3 = getind(nextln(line2))
          k1 = getind(line1)
          k2 = getind(line2)
```

```
        call relink(k0, k3, k0, k3)
        if (line3 > line1) {
            curln = line3
            line3 = line3 - (line2 - line1 + 1)
            }
        else
            curln = line3 + (line2 — line1 + 1)
        k4 = getind(line3)
        k5 = getind(nextln(line3))
        call relink(k4, k1, k2, k5)
        call relink(k2, k5, k4, k1)
        move = OK
        }
    return
    end
```

Gyakorlatok

6.7 Miért hívtuk a **move**-ban **getind**-et és **relink**-et olyan furcsa sorrendben? Mi történhet, ha először elintézzük az összes **getind** hívásokat, majd az összes **relink** hívásokat? Mi történhet, ha **k3**-at **k1** és **k2** után számítjuk ki? A **relink** első hívása miért nem áll közvetlenül **k3** kiszámítása után? □

6.8 Vázlatos szerkesztőnket egészítsük ki a fenti parancsokkal, majd próbáljuk ki őket. Fordítsunk különös figyelmet a határesetekre, mint pl. **0i, .m.—1** és **1,$d.** □

6.9 A törölt sorok által elfoglalt tárhely visszanyerésére nem tettünk kísérletet. Milyen problémát jelentene ez? Hogyan valósítanánk meg egy hulladékgyűjtő rendszert a **delete** lehető legkisebb módosításával? (A **delete** az egyetlen szubrutin, amelyben tárveszteség van; a **c, d** és az **s** parancsok hívják.) Mi célszerűbb? Az összeszedett hulladéktárat felhasználni, amint lehet, vagy csak akkor, ha szükség van rá? Milyen méréseket kellene elvégezni álláspontunk igazolására? □

6.10 Valósítsuk meg a **k** másolóparancsot: a

(.,.)k *sor3* **p**

egy sorcsoportot áthelyezés helyett átmásol *sor3* után. □

6.6 A HELYETTESÍTŐ PARANCS

Eddig teljes szövegsorokkal foglalkoztunk, és sok szerkesztőprogram mást nem is tesz lehetővé. Nekünk azonban itt van az 5. fejezetből származó **change** program, melyre támaszkodva könnyen létrehozhatjuk az **s** *helyettesítő parancsot.* Ez szelektív módon helyettesít egy szövegmintához illeszkedő szövegrészt, és közelebb áll az emberi szemléletmódhoz, mintha kizárólag teljes sorokat lehetne helyettesíteni vagy a karakterpozíciókat csak oszlopszámokkal lehetne megadni. A legtöbb esetben kényelmesebbnek tűnik a szövegben sor-szinten mozogni (természetesen szövegkörnyezet alapján), majd a helyettesítések során karakterszintre leereszkedni. A *helyettesítő parancs* formája a következő:

(.,.) s /*minta*/*új*/**g,**

ahol a / elhatároló helyett bármely, az új sortól különböző karakter használható. A **g** utótagot akkor használjuk, ha — mint a **change**-nél — az összes illeszkedő

részsorozatot meg akarjuk változtatni; ha a **g** nem szerepel, akkor csak a bal oldali első sikeres illesztésnél kerül sor változtatásra.

A **docmd**-ben a helyettesítő parancsra vonatkozó programrész a következő:

```
else if (lin(i) == SUBSTITUTE) {
    i = i + 1
    if (optpat(lin, i) == OK)
      andif (getrhs(lin, i, sub, gflag) == OK)
      andif (ckp(lin, i + 1, pflag, status) == OK)
      andif (defalt(curln, curln, status) == OK)
          status = subst(sub, gflag)
    }
```

A **getnum**-hoz hasonlóan, itt is felhasználjuk az **optpat**-ot a minta kódolása céljából, majd a helyettesítő sorozat kódolása, és a **g** jelenlétének eldöntése érdekében a **getrhs** hívása következik. A **getrhs** viszont a **maksub**-ot használja, amelyet a **change** számára írtunk, ez végzi el a munka legnagyobb részét. Mind a **makpat**, mind a **maksub** rutint úgy írtuk meg, hogy az elhatároló tetszőleges jel lehet. Ez megengedi, hogy a helyettesítő utasításokban elhatárolóként bármely megfelelő karaktert használhassunk, nem csupán a törtvonalat, amelyet legtöbb példánkban alkalmaztunk. (Ez jól jön, ha törtvonalakat akarunk helyettesíteni, és nem szeretnénk minden alkalommal felhívójelet használni.) A **ckp** most is az esetleges parancsvégi **p**-t vizsgálja, amely az eredménysort nyomtatja ki, a **defalt** pedig szükség esetén beállítja a sorszámok alapfeltételezését.

```
# getrhs — helyettesítő sorozat átvétele az "s" parancshoz
   integer function getrhs(lin, i, sub, gflag)
   character lin(MAXLINE), sub(MAXPAT)
   integer maksub
   integer gflag, i

   getrhs = ERR
   if (lin(i) == EOS)
       return
   if (lin(i + 1) == EOS)
       return
   i = maksub(lin, i + 1, lin(i), sub)
   if (i == ERR)
       return
   if (lin(i + 1) == GLOBAL) {
       i = i + 1
       gflag = YES
       }
   else
       gflag = NO
   getrhs = OK
   return
   end
```

Most már nem maradt más hátra, mint a **subst** programja, melyet az 5. fejezetbeli **change** főrutinja után mintáztunk.

```
# subst = "sub" helyettesítése a minta helyére
    integer function subst(sub, gflag)
    character new(MAXLINE), sub(MAXPAT)
    integer addset, amatch, gettxt, inject
    integer gflag, j, junk, k, lastm, line, m, status, subbed
    include clines
    include cpat
    include ctxt

    subst = ERR
    if (line1 <= 0)
        return
    for (line = line1; line<= line2; line = line + 1) {
        j = 1
        subbed = NO
        junk = gettxt(line)
        lastm = 0
        for (k = 1; txt(k) ¬= EOS; ) {
            if (gflag == YES | subbed == NO)
                m = amatch(txt, k, pat)
            else
                m = 0
            if (m > 0 & lastm ¬=m) {    # az illesztett szöveg cseréje
                subbed = YES
                call catsub(txt, k, m, sub, new, j, MAXLINE)
                lastm = m
                }
            if (m == 0 | m == k) {      # nincs, vagy nulla
                                        # hosszúságú illeszkedés
                junk = addset(txt(k), new, j, MAXLINE)
                k = k + 1
                }
            else                    # az illesztett szöveg átlépése
                k = m
            }
        if (subbed == YES) {
            if (addset(EOS, new, j, MAXLINE) == NO) {
                subst = ERR
                break
                }
            call delete(line, line, status) # a pontról sem feledkezik meg
            subst = inject(new)
            if (subst == ERR)
                break
            subst = OK
            }
        }
    return
    end
```

A helyettesítő parancsnál hibának tekintjük, ha helyettesítésre egyáltalán nem kerül sor, mert ez gyakran azt mutatja, hogy nem a megfelelő mintát írtuk le, vagy hogy a mintát rossz sorra alkalmaztuk.

Gyakorlatok

6.11 Mi a jelentése a // egyes előfordulásainak az

/abc/;//;//s///

parancsban? Mi a parancs hatása? □

6.12 Mi történik, ha egy sor összes karaktereit töröljük, mint pl. a

s/?*//

parancs esetén? Mi történik, ha ezután az újsor-karaktert is töröljük a

s/@n//

paranccsal? Mi történik, ha egy nem üres sor végéről töröljük az újsor-karaktert? Mi történik, ha helyettesítéssel további újsor-karaktereket hozunk létre? Foglaljuk össze tömören ezt a viselkedést egy vagy két szabályba! □

6.13 Dolgozzuk ki az *átkódoló parancsot* **(.,.)t**-t, amely a 2. fejezetben szereplő **translit** programhoz hasonlóan az egyik jelkészletet egy másikra képezi le. Pl.

1,$t/a—z/A—Z/

a kisbetűket minden sorban nagybetűkre alakítja át. Miért lehet szükségünk a **translit** mellett erre a lehetőségre? E lehetőség birtokában miért lehet szükségünk a **translit**-ra? □

6.14 Hogyan valósítanánk meg az **u** *„csináld vissza”* utasítást, amely hatástalanítaná az utolsó helyettesítő parancsot (azaz az új sort a régivel helyettesítené)? Kiterjeszthető-e ez más parancsokra is? □

6.7 BEMENET/KIMENET

Már akkor is meglehetősen sokoldalú szerkesztőprogramhoz jutottunk volna, ha a tárgyalást itt abbahagynánk. Továbbá, egy kis többletmunkával olyan program birtokába is juthatnánk, melynek hívása

edit *file*

alakú, és amely feldolgozás előtt a *file*-t beolvassa a belső pufferjébe, majd kilépés előtt visszaírja azt. Mi azonban ehelyett néhány további parancs megvalósításával, amelyek file-ok explicit olvasását és írását teszik lehetővé, elérjük, hogy szelektív módon tudjunk file-okat egyesíteni, szétdarabolni, vagy több példányban lemásolni anélkül, hogy kilépnénk a szerkesztőből. Ezek a többletparancsok nagymértékben megkönnyítik a „kivágás és beragasztás” jellegű szerkesztést. Például, könyvünk kéziratának szerkesztésekor kiterjedten használtuk e parancsok mindegyikét abból a célból, hogy a programokat kipróbálás céljából elkülönítsük, majd javított változataikat visszaillesszük.

Az **e** *file* belépőparancs törli a belső puffert és átmásolja bele a *file*-t. Van egy **(.)r** *file* alakú olvasóparancs is, amely a file tartalmát közvetlenül az adott sorhoz illeszti a puffer tartalmának megváltoztatása nélkül, mintha egy **a** parancs után került volna leírásra. File-ok létrehozását vagy átírását az **(1,$)w** *file* alakú íróparancs végzi, amely sorok adott intervallumát másolja át a *file*-ba, átírva annak előző tar-

talmát. A parancs alapfeltételezése a teljes puffer kiírása, ha nincsenek sorszámok megadva, illetve egy sor kiírása, ha csak egyetlen sorszám van. Az **e** és az **r** parancs után a pont az utolsó olvasott sorra mutat. A **w** parancs nem változtatja meg a pont értékét és a puffer tartalmát, így a szerkesztés zavarása nélkül készíthetjük el a file közbenső változatait.

A file nevét a program az **edit** *file* paraméteréből, az első, file-nevet tartalmazó **r** vagy **w** parancsból, vagy a legutóbbi **e** parancsból veszi. A *file* paraméter nélküli B/K parancs ezt a megjegyzett nevet használja, ezért egy önmagában álló **e**, **r** vagy **w** parancs arra a file-ra hivatkozik, amellyel a munkát kezdtük. Az **f** parancsnak csupán az a feladata, hogy kinyomtatja a megjegyzett nevet, vagy beállítja azt, ha a parancsot **f** *file* formában használjuk. A megjegyzett file-név tárolására a **cfile** közös adatmezőt vezetjük be.

```
common /cfile/ savfil(MAXLINE)
    character savfil      # a megjegyzett file-név
```

A **docmd** hátralevő része:

```
else if (lin(i) == ENTER) {
    if (nlines == 0)
      andif (getfn(lin, i, file) == OK) {
          call scopy(file, 1, savfil, 1)
          call clrbuf
          call setbuf
          status = doread(0, file)
          }
    }
else if (lin(i) == PRINTFIL) {
    if (nlines == 0)
      andif (getfn(lin, i, file) == OK) {
          call scopy(file, 1, savfil, 1)
          call putlin(savfil, STDOUT)
          call putc(NEWLINE)
          status = OK
          }
    }
else if (lin(i) == READCOM) {
    if (getfn(lin, i, file) == OK)
          status = doread(line2, file)
    }
else if (lin(i) == WRITECOM) {
    if (getfn(lin, i, file) == OK)
      andif (defalt(1, lastln, status) == OK)
          status = dowrit(line1, line2, file)
    }
```

A **w** parancshoz tartozó **(1,$)** sorszámokra vonatkozó alapfeltételezést a korábban bemutatott **defalt** biztosítja.

A file-nevek átvételét és ellenőrzését **getfn** végzi, amely vagy nem kap paramétert, vagy legalább egy szóközből és egy file-névből álló paraméter feldolgozására van felkészülve.

```
# getfn — a file-név átvétele lin(i)-től
    integer function getfn(lin, i, file)
    character lin(MAXLINE,) file(MAXLINE)
    integer i, j, k
    include cfile

    getfn = ERR
    if (lin(i + 1) == BLANK) {
        j = i + 2          # az új file-név átvétele
        call skipbl(lin, j)
        for (k = 1; lin(j) ¬= NEWLINE; k = k + 1) {
            file(k) = lin(j)
            j = j + 1
            }
        file(k) = EOS
        if (k > 1)
            getfn = OK
        }
    else if (lin(i + 1) == NEWLINE & savfil(1) ¬= EOS) {
        call scopy(savfil, 1, file, 1)        # vagy a régi file-névé
        getfn = OK
        }
    # egyébként hiba
    if (getfn == OK & savfil(1) == EOS)
        call scopy(file, 1, savfil, 1)     # tárolás, ha nincs régi file-név
    return
    end
```

A **doread** és a **dowrit** függvények kinyomtatják az átvitt sorok számát, részint ellenőrzési célból, részint a művelet befejezésének jelzéséül. Ez a stratégia ellentmond a túl sokat beszélő programokról korábban kifejtett nézeteinknek, de azt tapasztaltuk, hogy a felhasználók szívesen látnak valami visszajelzést az olyan műveletekről, mint az **r** és **w**, melyek lényegesen megváltoztatják a lokális vagy a külső file állapotát. A sorszámlálás némileg megerősíti, hogy az átvitelre szánt információ valóban a helyére került. Az a tervezési elv, hogy „kerüljük a bőbeszédűséget", az ésszerűség határain belül mérlegelendő, nem pedig vakon követendő szabály.

```
# doread — "file" olvasása "line"-től kezdve
    integer function doread(line, file)
    character file(MAXLINE), lin(MAXLINE)
    integer getlin, inject, open
    integer count, fd, line
    include clines

    fd = open(file, READ)
    if (fd == ERR)
```

```
        doread = ERR
    else {
        curln = line
        doread = OK
        for (count = 0; getlin(lin, fd) ¬= EOF; count = count + 1) {
            doread = inject(lin)
            if (doread == ERR)
                break
            }
        call close(fd)
        call putdec(count, 1)
        call putc(NEWLINE)
        }
    return
    end

# dowrit — "from"-tól "to"-ig terjedő szöveg írása file-be
    integer function dowrit(from, to, file)
    character file(MAXLINE)
    integer create, gettxt
    integer fd, from, k, line, to
    include ctxt

    fd = create(file, WRITE)
    if (fd == ERR)
        dowrit = ERR
    else {
        for (line = from; line <= to; line = line + 1) {
            k = gettxt(line)
            call putlin(txt, fd)
            }
        call close(fd)
        call putdec(to—from+1, 1)
        call putc(NEWLINE)
        dowrit = OK
        }
    return
    end
```

Gyakorlatok

6.15 Az Olvasó egészítse ki szerkesztőjét B/K parancsokkal, majd próbálja ki azokat. Ha a használt rendszer nem engedi meg file-ok név szerinti megadását, keressünk megoldást a 3. fejezetben szereplő anyag alapján. □
6.16 Módosítsuk úgy az **r** és **w** parancsokat, hogy megerősítésként ne adjanak sorszámláló értéket. Végezzünk kísérleteket mind a két változattal! Melyiket részesítenénk előnyben? □
6.17 Az

1,$d
r *file*

parancssorozat legalább két fontos szempontból különbözik az

e *file*

parancstól. Melyek ezek? Érdemes-e a **d** parancsot úgy módosítani, hogy rendelkezzék **e** előnyeinek egy részével? □

6.8 GLOBÁLIS PARANCSOK

Ezzel befejeztük a szerkesztőprogram kidolgozását, kivéve a **g** globális előtag részleteit, amely, mint már láttuk, a következő formájú:

(1,$) g/*minta*/*parancs*

A sorszámok alapfeltételezése megegyezik a **w** parancséval: ha nem adunk meg sorszámot, a teljes puffer értendő. A *parancs* a *mintához* illeszkedő minden sorra végrehajtódik. Egyes esetekben kényelmesebb azt a mintát megadni, amely a változatlanul maradó sorokhoz illeszkedik, ezért definiáljuk **g** komplemensét, **x**-et:

(1,$)x/*minta*/*parancs*

amely végrehajtja a *parancs*ot minden soron, amelyik *nem* tartalmazza a mintát. (A programot szimbolikus konstansok felhasználásával írtuk, melyek közül csupán egyet kell megváltoztatni, ha a mi választásunk helyett az Olvasó mást részesít előnyben.)

A *parancs* bármilyen szerkesztőparancs lehet, kivéve az **a**, **c** és **i** parancsokat, melyek globális változatait a 6.25 gyakorlatban szerepeltetjük. A *parancs* előtt szövegkörnyezet alapján való keresést tartalmazó sorszámok stb. is állhatnak. Pl.:

g/%#/p

kinyomtatja a #-val kezdődő sorokat, mint pl. a mi szubrutinjaink bevezető sorait. Vagy a

g/%#/.,.+1p

kinyomtatja a megjegyzés sort és a rákövetkező sort, amely utóbbi rendszerint a szubrutin vagy függvény deklarációjának bevezető sora.

Mivel a globális előtag után megengedjük a **d**, **m** és **r** parancsokat is, egy parancs végrehajtása a pufferban levő sorok mindenféle átrendeződését hozhatja létre, ezért pontosan meg kell határoznunk a sorok vizsgálatának és a rajtuk végzett műveletek végrehajtásának sorrendjét. Gondoskodnunk kell arról is, hogy a szerkesztő ne kerüljön végtelen ciklusokba és többé-kevésbé azt tegye, amit akarunk és elvárunk tőle. Az általunk kiválasztott módszer talán nem tökéletes, de működik és egyszerű. Először végigmegyünk a teljes anyagon, megjelölve a mintához illeszkedő **(g)**, illetve a nem illeszkedő **(x)** sorokat. Ezzel egyidejűleg a sorokon levő minden más jelet törlünk. Mindezt a **ckglob** rutin végzi, amelynek feladata a globális előtag feldolgozása, ha ilyen van.

```
# ckglob — az érintett sorok megjelölése globális előtag esetén
   integer function ckglob(lin, i, status)
   character lin(MAXLINE)
   integer defalt, getind, gettxt, match, nextln, optpat
   integer gflag, i, k, line, status
   include cbuf
   include clines
   include cpat
   include ctxt

   if (lin(i) ¬= GLOBAL & lin(i) ¬= EXCLUDE)
      status = EOF
   else {
      if (lin(i) == GLOBAL)
         gflag = YES
      else
         gflag = NO
      i = i + 1
      if (optpat(lin, i) == ERR | defalt(1, lastln, status) == ERR)
         status = ERR
      else {
         i = i + 1
         for (line = line1; line <= line2; line = line + 1) {
            k = gettxt(line)
            if (match(txt, pat) == gflag)
               buf(k+MARK) = YES
            else
               buf(k+MARK) = NO
            }
         for (line = nextln(line2); line ¬= line1; line = nextln(line)){
            k = getind(line)
            buf(k+MARK) = NO
            }
         status =OK
         }
      }
   ckglob = status
   return
   end
```

A munka hátralevő részét a **doglob** végzi, amelyet a főprogramból akkor hívunk, ha **ckglob** érvényes globális előtagot talál. A **doglob** az első sortól kezdve végigvizsgálja a sorokat, hogy meg vannak-e jelölve. Ha **g** jelű sort talál, törli a jelölést, beállítja a **curln**-t, és **docmd** hívásával végrehajtja a parancsot. Különben továbbhalad a pufferban, gondosan számolva, hogy hány sort lépett át az utolsó **g** jel óta. Amikor egy újabb jelzés felfedezése nélkül a puffer végére ér **(count> lastln)**, akkor munkáját befejezte.

```
# doglob — a lin(i)-nél levő parancs végrehajtása a jelölt sorokon
   integer function doglob(lin, i, cursav, status)
   character lin(MAXLINE)
   integer docmd, getind, getlst, nextln
   integer count, cursav, i, istart, k, line, status
   include cbuf
   include clines

   status = OK
   count = 0
   line = line1
   istart = i
   repeat {
      k = getind(line)
      if (buf(k+MARK) == YES) {
         buf(k+MARK) = NO
         curln = line
         cursav = curln
         i = istart
         if (getlst(lin, i, status) == OK)
           andif (docmd(lin, i, YES, status) == OK)
               count = 0
         }
      else {
         line = nextln(line)
         count = count + 1
         }
      } until (count > lastln | status ¬=OK)
   doglob = status
   return
   end
```

Minden egyes megjelölt sorra ráállítjuk a pont értékét, majd **docmd**-vel végrehajtatjuk a parancsot. (A **YES** paraméter azt mutatja, hogy a parancs végrehajtása a globális előtag vezérlése alatt megy végbe. Ez csak az **a**, **c**, **i** és **q** parancsok esetében befolyásolja a program munkáját.) A parancs maga módosíthatja a pont értékét, hozzáférhet több sorhoz is stb. Pl. a

```
g/subroutine/.,/[ @t] * end$/p
```

parancs kinyomtattatja az összes szubrutint; valahányszor egy **subroutine** szót tartalmazó sort talál, az **end** utasításig terjedő összes sort kiíratja. Bonyolultabb példaként tekintsük a következőt:

```
g/%/m0
```

Ez a parancs minden sort megjelöl, majd elölről kezdve minden sort a puffer elejére visz át. Végeredményben fordított sorrendben kapjuk a sorokat. Ugyanezen a műveleten alapuló példa, amelyet rendszeresen alkalmazunk is:

```
g/valami/m0
0;\\=
```

Ez a két parancs minden „valami”-t tartalmazó sort átvisz a puffer elejére, majd megkeresi az utolsó ugyanilyent. Végeredményben megszámlálja a „valami”-k előfordulását a puffer összekeverése árán. Minták előfordulását megszámlálhatjuk az

```
x/minta/d
$=
```

segítségével, ha nem zavaró, hogy a pufferből sorok törlődnek.

Gyakorlatok

6.18 Az alábbi művelet megkísérli a pufferben levő páros és páratlan sorszámú sorok szétválasztását, de nem működik helyesen:

```
g/%/.+1m$
```

Mit csinál ez tulajdonképpen? Hogy kellene megváltoztatni a **move**-ot, hogy ez a művelet is helyesen működjön (a többi hasznos művelet pedig ne romoljon el)? □

6.19 Bizonyítsuk be, hogy a **doglob** nem eshet végtelen ciklusba! □

6.20 Hogyan lehetne megjavítani a **g** feldolgozásának hatékonyságát? Megéri-e a munkát? □

6.9 A FŐPROGRAM

Most végre bemutathatjuk az **edit** főprogramját. Mielőtt azonban ezt megtennénk, közöljük a **docmd** teljes szövegét, hogy az Olvasó egy tagban lássa és felfrissíthesse az emlékezetét. Mint már említettük, a program bár hosszú, csupán egyetlen **case** utasításból áll, amely a sok lehetőség közül választ ki egyet.

```
# docmd — parancsfeldolgozás a globálisok kivételével
   integer function docmd(lin, i, glob, status)
   character file(MAXLINE), lin(MAXLINE), sub(MAXPAT)
   integer append, delete, doprint, doread, dowrit, move, subst
   integer ckp, defalt, getfn, getone, getrhs, nextln, optpat, prevln
   integer gflag, glob, i, line3, pflag, status
   include cfile
   include clines
   include cpat

   pflag = NO                # d, m vagy s állíthatja
   status = ERR
   if (lin(i) == APPENDCOM) {
      if (lin(i + 1) == NEWLINE)
            status = append(line2, glob)
      }
```

```
else if (lin(i) == CHANGE) {
     if (lin(i + 1) == NEWLINE)
        andif (defalt(curln, curln, status) == OK)
        andif (delete(line1, line2, status) == OK)
              status = append(prevln(line1), glob)
     }
else if (lin(i) == DELCOM) {
     if (ckp(lin, i + 1, pflag, status) == OK)
        andif (defalt(curln, curln, status) == OK)
        andif (delete(line1, line2, status) == OK)
        andif (nextln(curln) ¬= 0)
              curln = nextln(curln)
     }
else if (lin(i) == INSERT) {
     if (lin(i + 1) == NEWLINE)
              status = append(prevln(line2), glob)
     }
else if (lin(i) == PRINTCUR) {
     if (ckp(lin, i + 1, pflag, status) == OK) {
              call putdec(line2, 1)
              call putc(NEWLINE)
              }
     }
else if (lin(i) == MOVECOM) {
     i = i + 1
     if (getone(lin, i, line3, status) == EOF)
              status = ERR
     if (status == OK)
        andif (ckp(lin, i, pflag, status) == OK)
        andif (defalt(curln, curln, status) == OK)
              status = move(line3)
     }
else if (lin(i) == SUBSTITUTE) {
     i = i + 1
     if (optpat(lin, i) == OK)
        andif (getrhs(lin, i, sub, gflag) == OK)
        andif (ckp(lin, i + 1, pflag, status)  == OK)
        andif (defalt(curln, curln, status) == OK)
              status = subst(sub, gflag)
     }
else if (lin(i) == ENTER) {
     if (nlines == 0)
        andif (getfn(lin, i, file) == OK) {
              call scopy(file, 1, savfil, 1)
              call clrbuf
              call setbuf
              status = doread(0, file)
              }
     }
```

```
else if (lin(i) == PRINTFIL) {
    if (nlines == 0)
      andif (getfn(lin, i, file) == OK) {
          call scopy(file, 1, savfil, 1)
          call putlin(savfil, STDOUT)
          call putc(NEWLINE)
          status = OK
          }
    }
else if (lin(i) == READCOM) {
    if (getfn(lin, i, file) == OK)
          status = doread(line2, file)
    }
else if (lin(i) == WRITECOM) {
    if (getfn(lin, i, file) == OK)
      andif (defalt(1, lastln, status) == OK)
          status = dowrit(line1, line2, file)
    }
else if (lin(i) == PRINT) {
    if (lin(i + 1) == NEWLINE)
      andif (defalt(curln, curln, status) == OK)
          status = doprnt(line1, line2)
    }
else if (lin(i) == NEWLINE) {
    if (nlines == 0)
          line2 = nextln(curln)
    status = doprnt(line2, line2)
    }
else if (lin(i) == QUIT) {
    if (lin(i + 1) == NEWLINE & nlines == 0 & glob == NO)
          status = EOF
    }
# különben status = ERR
if (status == OK & pflag == YES)
    status = doprnt(curln, curln)
docmd = status
return
end
```

Most az **edit** főprogramja következik, deklarációival együtt. Ez a program végzi az **edit** *file* hívóparancsban szereplő elhagyható file-név feldolgozását is.

```
# edit — főprogram
    character lin(MAXLINE)
    integer ckglob, docmd, doglob, doread, getarg, getlin, getlst
    integer cursav, i, status
    include cfile
    include clines
    include cpat

    call setbuf
    pat(1) = EOS
    savfil(1) = EOS
    if (getarg(1, savfil, MAXLINE) ¬= EOF)
        if (doread(0, savfil) == ERR)
            call remark("?.")
    while (getlin(lin, STDIN) ¬= EOF) {
        i = 1
        cursav = curln
        if (getlst(lin, i, status) == OK) {
            if (ckglob(lin, i, status) == OK)
                status = doglob(lin, i, cursav, status)
            else if (status ¬= ERR)
                status = docmd(lin, i, NO, status)
            # különben hiba, nincs mit tenni
            }
        if (status == ERR) {
            call remark("?.")
            curln = cursav
            }
        else if (status == EOF)
            break
        # különben OK, a ciklus folytatható
        }
    call clrbuf
    stop
    end
```

Végül bemutatjuk az **edit** szubrutinjainak hierarchiáját. Amint ezt korábban is tettük, több alacsonyszintű kiszolgáló rutint elhagytunk, hogy a struktúra még kezelhető méretű maradjon.

```
edit
    setbuf
        relink
    doread
    getlst
        getone
            getnum
                optpat
                    makpat
                ptscan
                    gettxt, match
    ckglob
        optpat, match, gettxt, getind, defalt
    doglob
        getlst, docmd, getind
    docmd
        append
            inject
                getind, relink
        defalt
        delete
            getind, relink
        ckp
        getone
        move
            getind, relink
        optpat
        getrhs
            maksub
        subst
            gettxt, amatch, delete, inject
        getfn
        doread
            open, inject, close
        dowrit
            create, gettxt, putlin, close
        doprnt
            gettxt, putlin
    clrbuf
```

Bár a **docmd**-hez igen sok rutin tartozik, a helyzet mégsem olyan rossz, mint amilyennek látszik, mivel minden egyes hívásnál csak egyetlen esetre kell számítanunk. Ennek figyelembevételével az **edit** hierarchiája már tűrhetően áttekinthető.

Gyakorlatok

6.21 Az Olvasó fejezze be és alaposan próbálja ki saját szerkesztőprogramját! Célszerű lenne-e, ha a szerkesztő több tájékoztatást adna arról, amit csinál? Ha igen, módosítsuk az **append**-et úgy, hogy minden sor beolvasása előtt egy jelző *-ot nyomtasson ki, és minden parancsot kiírás kövessen. Próbaként hasz-

náljuk mindkét változatot egy ideig, majd ezután döntsük el, melyiket részesítjük előnyben. Szóba jöhet egy választást lehetővé tevő **o** parancs, amely a szerkesztő bőbeszédű (**ov**) vagy szűkszavú (**os**) használatát valósítja meg. Melyik állapotával induljon a program? □

6.22 Bővítsük a parancsok körét egy olyan újabb paranccsal, amely kikapcsolja a **?**, **[]** stb. metakarakterek külön jelentését. Ha a metakaraktereket kikapcsoljuk, lehetővé kell tennünk egyedi jelentésük ideiglenes visszaállítását oly módon, hogy a karakter elé egy felhívójelet írunk. Alapfeltételezésként a metakarakterek ki- vagy bekapcsolt állapotban legyenek? □

6.23 Valósítsuk meg az **l** listázó parancsot, amely a **p** nyomtató parancs, azzal a különbséggel, hogy a különben láthatatlan karakterek — mint pl. visszalépések, tabulátorok és ki nem írható karakterek — helyett azok valamilyen látható reprezentációját írja ki. □

6.24 Vannak akik ellenzik a puszta kérdőjel (?) mint egyetlen hibajelzés alkalmazását. Valósítsunk meg egy olyan "**?**" parancsot, amely felvilágosítást ad a legutolsó hibáról. □

6.25 Hogyan definiálnánk egy globális kiegészítő parancsot, hogy a beillesztésre (vagy cserére, vagy beszúrásra) kerülő szöveget csak egyszer kelljen bevinni? Miképpen lehetne ezt megvalósítani? Hogy lehetne egy globális előtag hatáskörét tetszőleges számú parancsra kiterjeszteni? Hogyan kezelnénk egymásba skatulyázott globális előtagokat? A rekurzió egyszerűsítené-e a feladatot? □

6.26 Ha az Olvasó által használt rendszer megköveteli vagy nyomatékosan javasolja, hogy a sorokhoz többé-kevésbé állandó sorszámok tartozzanak, illetve az Olvasó ezen sorszámok használatát esztétikai okokból helyesnek tartja, végezze el a szerkesztőben a kezelésükhöz szükséges módosítást. Az abszolút sorszámoknak használhatónak kell lenniük, ezeket az **a**, **c** és **i** parancsok generálják valamilyen módon, az **s** és **m** parancsok a sorszámokat megfelelően kezelik, az **r** és **w** parancsok pedig szintén foglalkoznak velük. □

6.10 MUNKAFILE-OK

A működő szerkesztőprogram birtokában figyelmünket most már annak tökéletesítésére összpontosíthatjuk. Először is a puffertárolóval foglalkozunk, amely a jelen változatban tagadhatatlanul primitív. Számottevő tárterületet takaríthatunk meg, ha csak a kapcsolási információt tartjuk az operatív tárban, és a terjedelmesebb szöveget egy munka-file-ban kezeljük.

A fő változás a szövegsorok szétválasztása a mutatóinformációtól, mivel nem lesz egyszerre a teljes szöveg a tárban. Legnagyobb része egy munkafile-ban helyezkedik el, egyelőre meghatározatlan elrendezésben. Az operatív tárban levő információt oly módon kell szervezni, hogy (néhány alapművelet felhasználásával) a munkafile-t, az operatív tár korlátlan kiterjesztéseként kezelhessük a hosszabb hozzáférési időtől eltekintve.

Tekintsünk egy helyettesítő parancsot. Hozzá kell férnünk a szövegsorhoz, ami azt jelenti, hogy meg kell azt találnunk a munkafile-on, hacsak nincs olyan szerencsénk, hogy az már az operatív tárban van. A sor jelenlegi változatát törölni kell a munkafile-ból, ami egyszerűen elintézhető úgy, hogy elfeledkezünk róla — a szöveg a munkafile-ban marad, de mutató nem mutat rá. Ezután be kell illeszteni a helyettesítő sort, amit legegyszerűbben a munkafile végéhez való csatlakoztatással oldhatunk meg. Ha mindezt elvégeztük, nyilván szükségünk lesz a mutató információra,

akárcsak a táron belüli változat esetében, hogy el tudjuk dönteni, melyik sor után melyik következik; tudnunk kell, hogy egy sor aktuális változatát hol találhatjuk meg a munkafile-ban, hogy be tudjuk olvasni; a hozzáférés érdekében a munkafile-ban el kell tudnunk jutni ehhez a ponthoz; az újabb változat kiírása céljából pedig el kell tudnunk jutni a munka-file végére is.

A **buf** tömb most is tartalmazza a **PREV** és **NEXT** mutatókat, valamint a **MARK**-ot. Mivel a **getind** és a **relink** csupán ezt az információt használja, ezek az új tervezéssel nem változnak meg. Azonban a **buf** minden elemében szükségünk lesz két újabb adatra: a sornak a munkafile-on belüli **SEEKADR** kezdőcímére, valamint **LENG**-re, a sor hosszára.

Ahelyett, hogy a munkafile-t valamilyen meghatározott tárolóeszközhöz, pl. mágnesszalaghoz vagy mágneslemezhez kötnénk, két további alapművelet bevezetésével hozzuk létre a kapcsolatot. Az egyik alapművelet a

```
seek(offset, fildes)
```

amely a megadott file-t olvasás vagy írás céljából **offset**-re pozicionálja. (A másik alapművelet neve **readf**; erre rövidesen visszatérünk.) A **fildes** a file belső neve, melyet az **open** vagy a **create** szolgáltat. Az **offset** kezelése során összeadjuk a **putlin** hívásakor kapott sorhosszakat, ily módon az **offset**-et használó **seek** a **putlin** n-edik hívása után a file-t közvetlenül a kiírt szöveg n-edik sora utánra állítja. Ez láthatóan karakter-orientált; olyan környezetben, ahol a sorok rekordokká válnak, a rekordok számát kell pozicionáló információként használnunk. Mivel a **seek** nagyon sokban függ az egyes rendszerek sajátosságaitól, a programot magát itt nem mutatjuk be.

Annyit elmondhatunk, hogy a **seek** elvégzi a szükséges visszacsévéléseket és előrelépéseket (mágnesszalag esetén) vagy kereső műveleteket (mágneslemez-file esetén) abból a célból, hogy megtaláljon egy korábban felírt szövegsort vagy előrehaladva eljusson a file végéig. Ezért a program egy file-t karakterek folytonos sorozatának tekinthet, és az alapműveletekre bízhatja az olyan egységekkel való foglalkozást, mint a rekordok és blokkok. Ez a célszerű tervezés.

Az **inject** sokban hasonlít a korábbi változathoz, attól eltekintve, hogy az alantas munka nagy részét egy új alárendelt modulra, a **maklin**-re bíztuk.

```
# inject (munkafile)—lin beillesztése curln után, írás a munkafile-ba
   integer function inject(lin)
   character lin(MAXLINE)
   integer getind, maklin, nextln
   integer i, k1, k2, k3
   include clines

   for (i = 1; lin(i) ¬= EOS; ) {
        i = maklin(lin, i, k3)
        if (i == ERR) {
             inject = ERR
             break
             }
```

```
          k1 = getind(curln)
          k2 = getind(nextln(curln))
          call relink(k1, k3, k3, k2)
          call relink(k3, k2, k1, k3)
          curln = curln + 1
          lastln = lastln + 1
          inject = OK
          }
     return
     end
```

Az új rutinjainkkal való kapcsolattartás nagy része magában **maklin**-ban összpontosul.

```
   # maklin(munkafile) — új sorhely kialakítása, a szöveg átmásolása
                              # a munkafile-ba
     integer function maklin(lin, i, newind)
     character lin(MAXLINE)
     integer addset, length
     integer i, j, junk, newind, txtend
     include cbuf
     include cscrat
     include ctxt

     maklin = ERR
     if (lastbf + BUFENT > MAXBUF)
          return          # nincs hely az új sor számára
     txtend = 1
     for (j = i; lin(j) ¬= EOS; ) {
          junk = addset(lin(j), txt, txtend, MAXLINE)
          j = j + 1
          if (lin(j - 1) == NEWLINE)
               break
          }
     if (addset(EOS, txt, txtend, MAXLINE) == NO)
          return
     call seek(scrend, scr)     #a sor csatlakoztatása a munkafile végéhez
     buf(lastbf + SEEKADR) = scrend
     buf(lastbf + LENG) = length(txt)
     call putlin(txt, scr)
     scrend = scrend + buf(lastbf + LENG)
     buf(lastbf + MARK) = NO
     newind = lastbf
     lastbf = lastbf + BUFENT
     maklin = j          # a következőként vizsgálandó karakter lin-ben
     return
     end
```

A file pillanatnyi végének pozíciója **scrend**, ahová az új szöveget csatlakoztatni lehet; az **scr** a munkafile belső neve, amelyet a **create** hozott létre. Ezek a változók a **cscrat** nevű közös adatmezőben helyezkednek el:

```
common /cscrat/ scr, scrend
    integer scr         # munkafile-azonosító
    integer scrend      # a munkafile-on levő információ vége
```

A **lastbf** a **buf**-ban használt utolsó elemet jelöli; **BUFENT** értéke 5, ez a **buf**-ban az egy sorra vonatkozó bejegyzés mérete. A **buf** és a **lastbf** a **cbuf** nevű közös adatmezőben helyezkedik el, amely az előző változathoz képest nem módosult, csupán az egy sorra vonatkozó bejegyzés szerkezete változott.

```
common /cbuf/ buf(MAXBUF), lastbf
    character buf       # a mutatók szerkezete minden sorra:
        # buf(k+0)  PREV       előző sor
        # buf(k+1)  NEXT       következő sor
        # buf(k+2)  MARK       globális parancsok jele
        # buf(k+3)  SEEKADR    hol van a sor a munkafile-ban
        # buf(k+4)  LENG       a munkafile-beli hossz
    integer lastbf      # a buf-ban használt utolsó mutató
```

A **gettxt** most is egy sornyi szöveget hoz be; mi úgy valósítottuk meg, hogy a **seek** segítségével megkeresi a munkafile megfelelő helyét, majd a szöveget beolvassa **txt**-be. Az olvasást a

```
readf(buffer, count, fildes)
```

alakú alapművelet végzi, amely **count** számú karaktert olvas **fildes**-ből a **buffer**-ba. Ez a második, a munkafile-ok kezelése céljából létrehozott alapművelet (**seek** volt az első). A **readf** ismét elegendően rendszerfüggő ahhoz, hogy megvalósításának ismertetésétől eltekintsünk.

```
# gettxt (munkafile) — "line"-hez tartozó szöveg átvitele txt-be
    integer function gettxt(line)
    integer getbuf, getind
    integer j, k, line
    include cbuf
    include cscrat
    include ctxt

    k = getind(line)
    call seek(buf(k + SEEKADR), scr)
    call readf(txt, buf(k + LENG), scr)
    j = buf(k + LENG) + 1
    txt(j) = EOS
    gettxt = k
    return
    end
```

Munkafile-t használó módszerünk úgy működik, hogy minden időpillanatban csak egy sornyi szöveget tart az operatív tárban. Ez nem nagyon hatékony és ezért egyszerre több sort is célszerű bent tartani. Az egyik gyakorlat a tár gazdaságosabb kihasználásával foglalkozik.

Miután a munkafile rutinok általános működését megismertük, hozzáfoghatunk a kezdőérték-beállító és a befejező rutinok elkészítéséhez. Rendszerint ebben a sorrendben a legcélszerűbb haladni; a kezdőérték-beállítást és a befejezést a végére hagyni, mivel csak ekkor tudjuk a teendőket megfelelő módon áttekinteni.

```
# setbuf (munkafile) — munkafile létrehozása, 0. sor beállítása
    subroutine setbuf
    integer create
    integer k
    include cbuf
    include clines
    include cscrat
    string scrfil "scratch"
    string null ""

    scr = create(scrfil, READWRITE)
    if (scr == ERR)
        call cant(scrfil)
    scrend = 0
    lastbf = LINE0
    call maklin(null, 1, k)      # üres 0. sor létrehozása
    call relink(k, k, k, k)      # a kezdeti kapcsolt lista létrehozása
    curln = 0
    lastln = 0
    return
    end
```

```
# clrbuf (munkafile) — munkafile megszüntetése
    subroutine clrbuf
    include cscrat
    string scrfil "scratch"

    call close(scr)
    call remove(scrfil)
    return
    end
```

Gyakorlatok

6.27 Helyettesítsük a meglevő puffervezérlő elemi műveleteket a munkafile-ra vonatkozó készlettel (beleértve a **seek**-et és a **readf**-et is), és lőjük be őket. File-jainkból vett mintán mérjük meg az átlagos sorhosszúságot (mely eszközöket használnánk ennek elvégzésére?), és ennek felhasználásával becsüljük meg a szerkesztő két változatának relatív kapacitását! □

6.28 Lássuk be, hogy ha a sorvégi újsor-karakter törlését nem engedjük meg, akkor a **buf**-ban nincs szükség a **LENG** adatra, és a **readf** elemi művelet **getlin**-nel helyettesíthető. □

6.29 Érdemes-e hulladékgyűjtést végezni a **buf**-beli használatlan mutató adatmezőkre? Próbáljuk ki! És a munkafile-ban felszabadított szövegterületre? □

6.30 A legtöbb rendszerben az **edit** lényegesen felgyorsítható, ha megpróbáljuk „kitalálni", mely sorok lesznek a közvetlen közeli jövőben felhasználva, és egy alkalommal több sort olvasunk be. Módosítsuk a **gettxt**-et vagy valamelyik alárendelt rutinját oly módon, hogy behívása esetén a **gettxt** egy sorcsoportot olvasson be, hacsak a sor nincs már az operatív tárban. Az izgalmas feladat olyan működési elv kialakítása, amely mellett a gyakori szerkesztési műveletek esetén a sor nagy valószínűséggel *már az operatív tárban legyen.* Melyek azok a gyakori műveletek, melyeket érdemes tökéletesíteni? A programnak mekkora részét kell ehhez megváltoztatni? Mennyire fogja ez felgyorsítani a különböző műveleteket? □

6.31 Mutassuk meg, hogy az **inject** által végzett munka n sornak a pufferba helyezésekor n^2-tel arányos. Végezzünk méréseket az **edit**-ben, hogy ez zavaró-e? Ha igen, hogyan lehetne **inject**-et lineáris eljárássá alakítani? □

6.32 A mágnesszalagnak megvan az a tulajdonsága, hogy az utoljára felírt rész utáni információ nem nyerhető róla vissza (ezért a szalag közepén nem lehet egyes darabokat újra írni). Jól működik-e munkafile karbantartó rendszerünk a mágnesszalagon? Miért célszerű lehetővé tenni, hogy a szövegszerkesztő mágnesszalaggal is működjön? Mik ennek a hátrányai? □

6.33 A tárkezelés egy másik lehetséges megvalósítása a szövegsorokra mutató pointereknek egy folytonos tömbben való elhelyezése. Ily módon kapcsolási információ helyett az n-edik sorra vonatkozó minden adat mindig a tömb n-edik pozíciójában helyezkedne el. A fentiek szerint tervezzük át a tárkezelést. Mely műveletek lettek egyszerűbbek, mint amilyenek a kapcsolt lista használata esetén voltak? Melyek lettek bonyolultabbak? □

6.34 Hogyan valósítanánk meg azt a szerkesztőt, amely minden információt egy munkafile-ban tart, a sorkapcsolási adatokat is beleértve? Tudunk-e ebben az esetben is mágnesszalagot használni? Érdemes-e hulladékgyűjtést végezni? □

6.35 Futtassuk a szerkesztőprogram különböző változatait egy azonos parancskészlettel, és hasonlítsuk össze a válaszidőket. Használjuk fel ezt az információt annak meghatározására, hogy mennyire érdemes a pufferkezelést finomítani? □

6.36 Ha az Olvasó rendszere tartalmaz egy **run** alapműveletet, amely lehetővé teszi egy parancs végrehajtását a futó programból, valósítsunk meg egy @ „átlépő" parancsot, amely megengedi, hogy a szerkesztőből az operációs rendszernek szóló parancsot adjunk ki. Például a

@ **edit** *file*

parancs az **edit** egy új példányát hívná be, amely a *file*-on dolgozik; amikor ez befejezte a munkáját, az aktuális szerkesztő folytatja a tevékenységét. Milyen módosításokra van szükség **setbuf**-ban és **clrbuf**-ban, hogy egymásba skatulyázott szerkesztőprogramok helyesen működjenek? □

*

A rendszer környezetének egyik, eddig nem említett szempontja, a külső világból érkező jelek, elsősorban a *megszakítások* kezelése. Egy szerkesztőnél szükség van arra, hogy a felhasználó le tudja állítani az éppen végrehajtás alatt levő parancsot, például meg tudjon szakítani egy hosszú ideig tartó nyomtatóparancsot anélkül, hogy információt veszítene. A megszakítások általában előre nem látható és többnyire kellemetlen időpontokban érkeznek, ezért a szerkesztőt fel kell készíteni azoknak

saját hatáskörében való kezelésére. Egy nyomtatási parancs esetén ez nem jelent túl nagy problémát, de gondoljunk azokra a bonyodalmakra, melyeket egy globális előtaggal ellátott átviteli parancs végrehajtása közben érkező megszakítás okoz. A feladat a folyamatban levő akció lehető legkorábbi leállítása, hogy a további szerkesztés megfelelő módon végrehajtható legyen.

*

6.37 Ha az Olvasó rendszere tartalmaz megszakítást érzékelő műveletet, módosítsa az **edit**-et úgy, hogy a megszakításokat helyesen kezelje, információveszteség, illetve a puffer tartalmának elrontása nélkül. Mit lehet csinálni, ha nincs módunk a megszakítás érzékelésére? □

*

Amikor egy file túl naggyá válik, az **edit** néhány hasznos szolgáltatásának végrehajtása komoly nehézségeket okozhat. Ilyenek pl. a globális parancsok és a \\-el való fordított irányú keresés. Alapvető fontosságú azonban, hogy a szerkesztő a feladatok legnagyobb részét nagy file-okon is végre tudja hajtani. Az egyik lehetőség az, hogy az **edit**-et durva áttervezéssel „másolószerkesztővé" alakítjuk, amely a standard inputot átmásolja a standard outputra és eközben szerkesztési műveletek korlátozott halmazát hajtja végre rajta. Ez a megoldás kizárja azokat a parancsokat, amelyek a file visszafelé haladó vagy ismételt elemzését igénylik; ezekhez tartozik a \\, a globális előtag néhány esete, az áthelyezési utasítás, valamint a sorszám-aritmetikában szereplő mínusz jel (például **/abc/−3**).

Az

/abc/, /def/ *parancs*

alakú parancs értelme a következő lenne: „Próbáld meg a parancsot végrehajtani az első, és minden további olyan soron, amely illeszkedik az első sorszámhoz mindaddig, amíg egy, a második sorszámhoz illeszkedő sort nem találsz; ezután kezdd el figyelni újra az első sorszámhoz való illeszkedést." Egyetlen sorszám használata azt jelentené, hogy a parancsot minden soron végre kell hajtani, amely illeszkedik a mintához; ha pedig sorszám egyáltalában nem szerepel, akkor minden soron. A többszörös parancsokat is meg kell engedni.

Ha van egy másolószerkesztő rutinunk **(sedit)**, akkor a **find** és a **change** rendre a következő két paranccsal fejezhető ki:

sedit/*minta***/p**

és

sedit/*minta***/s/** */helyettesítő/***gp**

Célszerű-e ennek dacára **find**-ot és **change**-t külön programokként megtartani?

*

6.38 Tervezzünk meg és dolgozzunk ki egy másolószerkesztőt a fent vázolt elképzelések alapján! □

6.11. ÖSSZEFOGLALÁS

Nehéz megfelelő nézőpontot találni egy olyan nagyméretű rendszer tervezésének vagy programozásának megítéléséhez, mint egy szövegszerkesztő, különösen, ha nincs némi tapasztalatunk annak használatában.

Hasznos lehet összeállítani egy listát a leggyakrabban előforduló szerkesztési feladatokról, majd összehasonlítani különböző szerkesztőket abból a szempontból, hogy hogyan oldják meg ezeket a feladatokat. Példaként egy olyan feladatot említünk, melyet gyakran kell elvégezni és mégis sok szerkesztő képességeit messze meghaladja. Ezen könyv szövegét számítógépünk file-rendszerében több, mint 100 file tartalmazza. Időről időre végig kell mennünk a teljes könyvön valamilyen módosítás elvégzése céljából, bárhol forduljon is az elő a file-ok halmazában. Ha minden file-t kézzel dolgoznánk fel, az elviselhetetlenül lassú lenne, és sok hibalehetőséget rejtene magában, ezért az **edit**-et a következőképpen használjuk:

Először készítünk egy forgatókönyvet, ami azon parancsok halmaza, melyeket minden file-on végre akarunk hajtani. (És természetesen ezt az **edit**-tel akarjuk elvégeztetni!) Egyik forgatókönyvünkben például az **ERROR** szót globálisan **ERR**-re változtattuk, hogy el tudjuk kerülni a kisbetűvel írott **error**-ral való összetévesztést. A forgatókönyv a következő volt:

```
f
g/ERROR/s//ERR/gp
w
```

Az **f** parancs a legrosszabbra is felkészülve kiadja a szerkesztés alatt álló file nevét; a globális előtaggal ellátott **s** parancs végrehajtja a cserét és kinyomtatja az érintett sorokat, végül a **w** parancs az új változatot visszaírja az eredeti file-ba.

Ezután lefuttatjuk a **listcat** („katalóguslistázó") programot, amely listát készít file-jainkról, soronként egy file névvel. A

```
listact >filelist
```

parancs a file nevek listáját a **filelist** nevű file-ba írja. (Emlékezzünk a > műveletre, melyet a 3. fejezetben tárgyaltunk.)

A következő lépésben a **filelist**-et szerkesztjük. A könyvünkhöz tartozó összes file nevében előfordulnak a **book** betűk, ezért először törlünk minden file nevet, amely nem tartozik könyvünkhöz a következő **x** paranccsal:

```
x/book/d
```

majd minden file nevet egy helyettesítés segítségével egy **e** paranccsá alakítunk:

```
1, $s/?*/e &/
```

Ennek eredményeképpen a puffer minden sora

e *file név*

alakú lesz. Ezután beolvassuk a forgatókönyv (script) egy példányát:

```
g/?/.r script
```

Ekkor a könyv eredeti file-jai már átalakultak egy-egy szerkesztő parancssorozattá, amelyek az **e** *file név* parancsból, valamint a forgatókönyv által előírt parancsokból állnak.

Most a teljes anyagot kiírjuk egy **command** nevű file-ba:

```
w command
q
```

Ezután az

```
edit <command
```

parancs hatására úgy hívódik be a szerkesztő, hogy bemenetét a **command** file-ból veszi. Ennek eredményeképpen **edit** sorban minden egyes file-ra végrehajtja a forgatókönyvet, bármit tartalmazzon is az.

A gyakorlatban ez a folyamat gyorsabb, mint ahogy le tudtuk írni. És természetesen, ha jól átgondoljuk, rájövünk, hogy a **command** file-t létrehozó szerkesztési műveletek egy másik file-ban is elhelyezhetők, melyet mondjuk **commandmaker**-nek nevezünk és valahányszor egy új parancsfile-ra van szükség, csak le kell futtatni ezt a műveletsorozatot. Más szerkesztőrendszerek esetén ez jó próba annak eldöntésére, hogy ezt a feladatot el tudják-e végezni ilyen egyszerűen.

Az **edit** parancsainak és lehetőségeinek összefoglalása következik. A szövegkörnyezet szerinti keresés során használt szövegminták, az **s** parancs, valamint a **g** és **x** előtagok összefoglalása az 5. fejezet végén található.

A sorszámok a következő komponensekből tevődnek össze:

17	*tizes számrendszerbeli egész szám*
.	*az aktuális sor*
$	*az utolsó sor*
/minta/	*szövegkörnyezet alapján való keresés előre*
\minta	*szövegkörnyezet alapján való keresés hátra*

A komponensek a + vagy − művelettel kombinálhatók:

.+1	*· és 1 összege*
.—1	*· és 2 különbsége*

A sorszámokat vesszőkkel vagy pontosvesszőkkel választjuk el egymástól; a pontosvessző a legutoljára előfordult sorszámot teszi meg aktuális sornak, mielőtt a szerkesztés folytatódna.

A parancsok előtt tetszőleges számú sorszám állhat (kivéve az **e**, **f** és **q** parancsokat, melyekben nem fordulhat elő sorszám). A szerkesztő az utolsó egy vagy két sorszámot szükség szerint használja fel. Ha két sorszámra van szükség, de csak egy van, akkor mindkettőt ez az egy helyettesíti. Ha sorszám nem szerepel, a következő alapfeltételezések valamelyike érvényes:

(.)	*használd az aktuális sort*
(.+1)	*használd a következő sort*
(.,.)	*használd az aktuális sort mindkét sorszám helyett*
(1,$)	*használd az összes sort*

A parancsok és a megfelelő sorszámra vonatkozó alapfeltételezések ábécé sorrendben a következők:

(.) **a**	*beillesztés a sor után (szöveg következik)*
(.,.) **c**	*szövegcsere (szöveg következik)*
(.,.) **dp**	*szövegtörlés*
e *file*	*áttérés a file-ra; minden előző szöveg elvész; a file nevét megjegyezzük*
f *file*	*a file név kinyomtatása és tárolása*
(.) **i**	*szövegbeszúrás a sor elé (szöveg következik)*
(.,.) **m** *sor3* **p**	*szöveg átvitele sor3 utánra*
(.,.) **p**	*szövegnyomtatás*
q	*kilépés a szerkesztőből*
(.) **r** *file*	*a file beolvasása és beillesztése a sor után*
(.,.) **s**/*minta*/*új*/**gp**	*a minta első előfordulásának az új-jal való helyettesítése (**g** a soron belüli összes előfordulás helyettesítését eredményezi)*
(1,$) **w** *file*	*a file kiírása (a pillanatnyi állapotot nem módosítja)*
(.) =**p**	*a sorszám kinyomtatása*
(.+1)	*egy sor kinyomtatása*

A parancs végén álló, elhagyható **p** az utolsó érintett sort íratja ki.

A globális előtagok a parancs ismételt végrehajtását eredményezik egy alkalommal minden illeszkedő sornál (**g**) vagy minden nem illeszkedő sornál (**x**)

(1,$)**g**/*minta*/*parancs*
(1,$)**x**/*minta*/*parancs*

A *parancs,* melyet a szokásos módon sorszámok előzhetnek meg, **a**, **c**, **i** és **q** kivételével bármi lehet. A pont a parancs végrehajtása előtt az utolsó illesztett sorra állítódik.

BIBLIOGRÁFIAI MEGJEGYZÉSEK

Az itt bemutatott szerkesztő legrégebbi, még kinyomozható változata a TECO. melyet az első PDP—1 időosztásos rendszerhez írtak a MIT-nél.* Ezt később L. P, Deutsch és B. W. Lampson megvalósította az SDS—940-en mint a QED „gyorsszerkesztőt"; lásd a *CACM* 1967. decemberi számában megjelent, „An online editor" c. cikket. A QED-t K. L. Thompson adaptálta a CTSS számára a MIT IBM 7090-esén, és később D. M. Ritchie megírta egy változatát a GE 635-re (most HIS—6070) a Bell Laboratóriumban.

Az utolsó változatot **ed**-nek hívják, a QED egy egyszerűsített formája a PDP—11-en, melyet Ritchie és Thompson írtak. A mi szerkesztőnk — legalábbis külső megjelenésében — nagyon hasonlít az **ed**-re.

Az A. van Dam és D. E. Rice által írt, „On-line text editing: a survey" c. cikk (Computing Surveys, 1971. szeptember) több más szerkesztőprogramot tárgyal.

* Massachussets Institute of Technology *(A fordító).*

7. fejezet

Formázás

Következő feladatunk egy szövegformázó megírása — azaz olyan programé, amely megfelelő sornyomtatón tetszetős formájú szöveg kiírására képes. Természetesen a „tetszetős formájú" és a „megfelelő" pontos jelentése igényeinktől és pénztárcánktól függ. A mi formázónk igen kis igényű, azokat a formázási lehetőségeket tartalmazza, amelyekről megfigyeltük, hogy a felhasználók ténylegesen használják nyomtatott szövegek előállításakor. Ez a formázó a kimenetét terminálon, sornyomtatókon és ezekhez hasonló eszközökön állítja elő, formázási lehetőségei pedig a sor végének automatikus kiegyenlítése, lapokra tagolás (átlépés a papír hajtásvonalán), lapszámozás, fejlécek felírása, középre állítás, aláhúzás, bekezdések és többféle sortávolság beállítása.

A formázó fontos eszköz mindenkinek, aki ír (beleértve a programjukat leíró programozókat), mivel ha az anyag egyszer már hibátlan, többé nem kell újragépelni. Ez, a nyilvánvaló költségmegtakarítás mellett valószínűbbé teszi azt is, hogy a hibák száma az idő múlásával csökken. A gépi formázás megkönnyíti a gépírási munkát, mivel a sorkezdet-beállítást, a szöveg középre állítását, az aláhúzást és más hasonló aprólékos műveletet a számítógép, és nem a gépíró végzi. Lehetővé teszi azt is, hogy az anyagon — a szöveg változtatása nélkül — alapvető formai módosításokat hajtsunk végre. De, ami talán a legfontosabb, úgy tűnik, hogy jobb minőségű írásművek előállítására ösztönöz, mivel a tökéletesítések keresztülvitele kevesebb többletmunkát igényel.

Némely esetben a hibátlanság az elsődleges cél. Például ezt a könyvet egy olyan fényszedőn hoztuk létre, amelyet az általunk megírandó formázónak egy igen sokat tudó „bátyja" vezérelt. A programok a szöveg részei. Bármely program kipróbálásakor egy szerkesztővel, amely a 6. fejezetben levőhöz hasonlít, leválasztjuk a program szövegét, lefordítjuk azt, és lefuttatjuk mindenféle kézi beavatkozás nélkül. Ezért meglehetősen biztosak vagyunk abban, hogy a szövegben szereplő programok azonosak azokkal, amelyeket mi teszteltünk.

A jelen fejezetben leírt **format** nevű formázóprogram meglehetősen hagyományos. A számára megadható input szöveg tartalmazza a kimenet alakját előíró formázóparancsokat is. Minden parancs egy pontból, egy kétbetűs névből és némely esetben valamilyen nem kötelezően használatos egyéb információból áll. Minden parancsnak a sor elején kell állnia oly módon, hogy a sor a parancson és paraméterein kívül mást ne tartalmazzon. Például, a

```
.ce
```

parancs a következő kimeneti sort középre állítja, a

.sp 3

parancs pedig három üres sort eredményez.

A legtöbb esetben a **format** felhasználójának nem kell törődnie a parancsokkal és a paraméterekkel; a formázás jórészt automatikus. Ez tulajdonképpen nem más, mint ésszerű, emberi szemléletmód. Elvileg egy formázási parancsot *egyáltalában nem tartalmazó* írásművet is elfogadható formában kellene kinyomtatnia. A paraméterek alapfeltételezéseit és a formázási műveleteket igyekeztünk kézenfekvően felépíteni úgy, hogy azok ne okozzanak meglepetéseket. Például, a kivitt sorokat — amennyire csak lehetséges — feltöltjük a szöveg szavaival a beolvasott sorok hosszától függetlenül. Az üres sorok új bekezdést hoznak létre. A szöveg lapokra tördelése szabályszerűen, az alsó és felső lapszél betartásával történik.

A programot ugyanakkor elég rugalmasra kell terveznünk, hogy az igényesebb felhasználások céljaira, magasabb szintű szolgáltatásokkal legyen kibővíthető. Magától értetődik, hogy a tájékozottabb felhasználóknak módot kell adni arra, hogy a paraméterek alapfeltételezéseit igényeiknek megfelelően megváltoztathassák. Azt is lehetővé kell tenni, hogy a meglevő parancsok felhasználásával a felhasználó új formázási műveleteket definiálhasson. A fenti lehetőségeket a fejezet végén levő gyakorlatokban vázoljuk fel.

7.1 PARANCSOK

Mint már említettük, minden parancs egy, a sor elején álló pontból — ami a szokásos szövegekben ritkán fordul elő — és egy kétbetűs névből áll. Tapasztalatunk szerint a felhasználók a legtöbb nyelv esetében a tömör parancsokat kedvelik, így a két betű elfogadható középút a tömörség és a mnemonikus érték között. Mindenesetre, a programot úgy írtuk meg, hogy egy ettől eltérő szempontot is kis változtatásokkal érvényre lehessen juttatni.

Alapfeltételezésként a **format** *feltölti* a kimeneti sorokat oly módon, hogy kinyomtatás előtt elhelyezi bennük a lehető legtöbb input szót. Továbbá a sorvégeket *kiegyenlíti* (így azok függőlegesen egymás alá esnek). Ezt úgy valósítja meg, hogy a kivitel előtt többletszóközöket szúr a sorba. A felhasználók általában feltöltött szöveget kívánnak, ezért választottuk ezt alapfeltételezésként. A feltöltés azonban letiltható, a *no-fill* parancs segítségével:

.nf

Ettől kezdve az input sorok átrendezés nélkül adódnak a kimenetre. A feltöltés a

.fi

fill paranccsal engedélyezhető újra.

Előfordulhat, hogy a **.nf** érkezésekor egy sor részlegesen feltöltött állapotban van, de kivitelre még nem került. A **.nf** mindenekelőtt kikényszeríti ezen sor kivitelét. Egy részben összeállított sor erőszakolt kivitelét megszakításnak *(break)* nevezzük. A megszakítás elve átszövi a **format**-ot; sok parancs eredményez implicit megszakítást. Ha explicit módon, például két bekezdés elválasztása céljából akarunk megszakítást létrehozni, a

.br

parancsot használhatjuk.

Természetesen előfordulhat, hogy a két bekezdés között egy üres sort is el akarunk helyezni. A

.sp

alakú *space* parancs egy megszakítást, majd egy üres sort hoz létre. Ha *n* számú üres sort akarunk kapni, használjuk a

.sp *n*

parancsot. (A parancs és paramétere között mindig van egy szóköz.) Ha az összes üres sor létrehozása előtt elérnénk a lap alját, a hátralevő üres sorok generálása letiltódik, így általában minden lap ugyanannál az első sornál kezdődik.

Alapfeltételezés szerint a sorok 1-es sorközzel követik egymást, de a sortávolság bármikor megváltoztatható: a

.ls *n*

parancs a sorközt *n*-re állítja ($n=2$ az ún. dupla sortávolság). A **.ls** parancs nem okoz megszakítást.

A **.bp** *(begin page)* parancs, hatására megszakítás jön létre, majd a kivitel egy új lap elején folytatódik. Ha a

.bp *n*

parancsot használjuk, a következő lap sorszáma *n* lesz. A lap aljánál előforduló **.bp** parancs hatástalan a lapszámozás esetleges átállításától eltekintve; üres lap nem keletkezik. Az aktuális laphosszúság megváltoztatható (megszakítás nélkül) a

.pl *n*

parancs segítségével.

A következő kimeneti sor középre állítását a *center* paranccsal valósíthatjuk meg:

.ce

a középre állítandó sor

A **.ce** megszakítást is eredményez. A következő *n* sor középre állítását a

.ce *n*

paranccsal írhatjuk elő. Ha nem akarjuk (vagy nem tudjuk) a sorokat pontosan megszámlálni a

.ce 1000

 sok, középre

 állítandó sor

.ce 0

parancsok hatására a két **.ce** parancs közötti sorok középre állítódnak. Középre állított soroknál nincs feltöltés.

Az *underlining* nagyrészt azonos a középre állítással: az

.ul *n*

parancs hatására a szöveg következő *n* sora aláhúzva kerül kivitelre. A **.ul** azonban nem okoz megszakítást, ezért a feltöltött szöveg szavait aláhúzhatjuk a következő módon:

szó, szó és
.ul
még több
szó

melynek eredményeképpen a következő sort kapjuk:

szó és szó és még több szó

A középre állítás és az aláhúzás tetszőleges sorrendben kombinálható: a

.ce
.ul
cím

parancsok középre állított és aláhúzott „*cím*"-et eredményeznek.

Az *indent* parancs a sorkezdet állítását vezérli: a

.in *n*

parancs hatására a további kimenetsorok *n* pozíciós bekezdési mélységgel íródnak ki. (A bekezdési mélység normális esetben 0.)

A *right margin* paranccsal a sor végét állíthatjuk be:

.rm *n*

a sorvégződést *n*-re állítja. A feltöltött sorok hossza a sorvégződés és a bekezdési mélység különbsége. Sem a **.in**, sem az **.rm** nem okoz megszakítást.

A hagyományos bekezdést a *temporary indent* parancs állítja elő:

.ti *n*

Ez megszakítást okoz, majd a bekezdési mélységet — kizárólag a következő sor számára — *n*-re állítja. Ha *n* kisebb, mint az aktuális bekezdési mélység, a sor visszafelé tolódik el („kilógó" bekezdés).

A minden lapon ismétlődő fej (header) és láb (footer) elhelyezését a **.he**, illetve a **.fo** parancsok végzik:

.he *ez lesz a lap tetején megjelenő (fej) szöveg*
.fo *ez lesz a lap alján megjelenő (láb) szöveg*

A kivitelre kerülő szöveg a parancs utáni első szóköztől különböző karakterrel kezdődik, de egy esetleges kezdő idézőjel nem jelenik meg a nyomtatási képen, ami lehetővé teszi, hogy a fej (láb) szóközökkel kezdődjék. Ha a fej (láb) szövege a # karaktert tartalmazza, ezt minden nyomtatáskor az aktuális lapsorszámmal helyettesíti. A **.he** és **.fo** nem okoz megszakítást.

Mivel a számok *abszolút* értékének használata gyakran kényelmetlen, a **format**-ban lehetőség van *relatív* (előjeles) értékek paraméterként való használatára is. Minden parancs, amelynek numerikus paramétere lehet, megengedi e helyett a $+n$ vagy $-n$ használatát, ami a pillanatnyi érték változtatásának nagyságát jelenti. Például: a

.rm —10
.in +10

parancsok a sor végének pillanatnyi értékét 10-zel csökkentik, illetve a bekezdési mélységet 10-zel növelik. Ennek értelmében a

.rm 10

és a

```
.rm +10
```

egészen mást jelent.

A relatív értékek különösen hasznosak a **.ti**, azaz az ideiglenes bekezdési mélység beállításakor: az

```
.in + 5
.ti + 5
```

parancsok a sorok elejét először 5 betűhellyel beljebb viszik, majd az első sort további 5 pozícióval beljebb kezdik. A

```
.in +5
.ti —5
```

parancsok pedig egy „kilógó" bekezdést eredményeznek, amilyeneket a számozott, pontokba szedett szövegeknél használunk:

1. Itt az ideje, hogy minden jó ember eljöjjön
a vendégségbe.

A szóközökkel kezdődő sorok különleges esetet jelentenek. Ha a sorban egyáltalán nincs szöveg, megszakítás következik be, és a pillanatnyi sortávolságnak megfelelő számú üres sor jön létre. Ezeket a sorokat — megjelenési helyüktől függetlenül — mindig figyelembe vesszük, ezért segítségükkel üres sorokat állíthatunk a lap tetejére is. Ha egy sor *n* szóközzel kezdődik, melyet szöveg követ, a sor megszakítást okoz és egy $+n$ ideiglenes bekezdési mélységet eredményez. Ezen speciális tevékenységek elősegítik azt, hogy egy formázási parancsokat nem tartalmazó szöveg is elfogadható külalakban jelenjen meg.

Az alábbiakban összefoglaljuk a parancsokat. Ha egy numerikus paraméter előtt + vagy – jel áll, *akkor az előző érték és a paraméter összege vagy különbsége* adja az új értéket, különben az új érték a paraméter értékével lesz azonos. Ha paraméter nincs megadva, akkor az alapfeltételezés szerinti értéke érvényes.

Parancs	**megszakítás?**	**alapfelt.**	**művelet**
.bp *n*	igen	$n=+1$	*n-nel számozott új lap kezdése*
.br	igen		*megszakítás*
.ce *n*	igen	$n=1$	*a következő n számú sor középre állítása*
.fi	igen		*sorfeltöltés elkezdése*
.fo	nem	üres	*a láb szövege*
.he	nem	üres	*a fej szövege*
.in *n*	nem	$n=0$	*bekezdési mélység=n szóköz*
.ls *n*	nem	$n=1$	*sortávolság=n*
.nf	igen		*sorfeltöltés leállítása*
.pl *n*	nem	$n=66$	*laphossz=n sor*
.rm *n*	nem	$n=60$	*sor vége=n*
.sp *n*	igen	$n=1$	*n számú üres sor kivitele*
.ti *n*	igen	$n=0$	*ideiglenes bekezdési mélység=n szóköz*
.ul *n*	nem	$n=1$	*a következő n sorban szereplő szavak aláhúzása*

Ez már eléggé gazdag készlet, de az Olvasónak minden bizonnyal további parancsok is eszébe jutnak. Ilyen lehetőségekre később még kitérünk.

Gyakorlatok

7.1 Határozzuk meg azt a kritériumot, melynek alapján el lehet dönteni, hogy mely parancsok okozzanak megszakítást és melyek ne. Egyik lehetőség az, hogy a parancsokat egy helyett két lehetséges karakter valamelyikével kezdjük. E szerint pl. **.sp** megszakítást okozna mint eddig, de **,sp** nem. Elemezzük ezt az elgondolást! □

7.2 Tervezzünk meg egy programcsatlakozást, amelynek segítségével egy szövegben megszámlálhatjuk a szavakat, a **format** parancsok kivételével!

7.2 A PROGRAM SZERKEZETE

Egy program fokozatos kiépítése sok előnnyel jár. Először elkészítjük azt a minimális programot, amely már valamire használható, majd ezen váz működőképességét megtartva, kis lépésekben haladunk tovább. Ily módon a nagy feladatot kisebb, és feltehetően könnyebben kezelhető részekre bontottuk. A váz tesztelése könnyebb, mivel az kisebb méretű. Ha a programot jól terveztük meg, akkor a későbbi beépített részek nincsenek szoros kapcsolatban a korábbiakkal, azért a legtöbb esetben önállóan tesztelhetők. Sikerélményünket is fokozza, ha az egészből *valami* igen hamar működőképes állapotba jut.

Nagyobb fejlesztések esetén önkéntes felhasználók vállalkozhatnak egy részben kész program korlátozott tevékenységeinek kipróbálására. Az ő véleményük létfontosságú visszacsatolást jelent mind a már meglevő, mind a még hátralevő részek értékelésének szempontjából. Gyakran kiderül, hogy a felhasználók igénye kisebb, mint azt eddig a fejlesztők közül bárki is gondolta volna, ezért a munka nehezének egy része későbbre halasztható. Kiderülhet az is, hogy a felhasználó egészen másra gondolt, mint a fejlesztők. Nem ésszerű a teljes rendszert magunkba zárkózva, egyes részek nyilvánosságra hozatala nélkül felépíteni. (A járatlan felhasználók botladozásai a programhibák között megfizethetetlenek; ők ugyanis úgy használják a programot, ahogy azt a fejlesztő álmában sem gondolta volna.)

A szövegformázóra mindez pontosan ráillik. Ha egyszer elkészült egy minimális lehetőségekkel rendelkező változat, a bővítések már úgy készíthetők el, hogy azok az előző programrészeket nem nagyon érintik (feltéve, hogy ésszerűen terveztük meg a rendszert). Miután azt már tisztáztuk, hogy milyen szolgáltatásokat kell annak nyújtania, a továbbiakban felvázoljuk a program fejlesztésének folyamatát.

A program szervezése szempontjából több lehetőségünk van, melyek közül kettő látszik ígéretesnek: Az egyik a bemenet szavankénti kezelése és a kimeneti sornak ezekből való összerakása olyan sor-orientált műveletek esetén, mint a *no-fill* és a *center* parancsok. A másik lehetőség a soronkénti bevitel és a sor szavakra bontása, feltöltött szöveg feldolgozása esetén. Ismerve a sorokon alapuló formázó műveletek, illetve parancsok számát — *no-fill, center* és *underlining* —, a soronkénti bevitel tűnik egyszerűbbnek. Ugyanakkor jó észtorna lehet a szavankénti kezelés részleteinek kidolgozása. Az Olvasó be fogja látni, hogy egyik szervezési mód sem ideális, mindegyiknek megvannak a maga hátrányai.

7.3 PARANCSELEMZÉS

A fő rutin soronként olvassa a bemenetet és szétválasztja a szövegsorokat, valamint a formázási parancsokat. Ennek programját minden további nélkül felírhatjuk:

```
# format — szövegformázó főprogram
   character inbuf(INSIZE)
   integer getlin

   call init
   while (getlin(inbuf, STDIN ¬= EOF)
      if (inbuf(1) == COMMAND)       # parancs
         call comand(inbuf)
      else                           # szöveg
         call text(inbuf)
   stop
   end
```

A **COMMAND** szimbolikus konstansnak a pont (.) jel felel meg, hacsak egy másikat nem részesítünk előnyben. Az **init** az összes paramétert az alapfeltételezés szerinti értékre állítja; elég érdektelen ahhoz, hogy bemutatásától itt eltekintsünk.

A részletezést a **command** rutinnal kezdjük, amely eldönti, hogy melyik utasításról van szó, mivel a parancsértelmező programok nagy részét megírhatjuk, mielőtt bármi máshoz fognánk. Átmenetileg a **text** olyan üres program lehet, amely csupán szövegsorokat másol át az **inbuf**-ból a standard output-ra, a **putlin** rutin segítségével.

A **command** szerkezetileg egy többutas elágazás a parancs típusa szerint, amelyet fokozatosan mutatunk be, és amelynek részleteire akkor térünk ki, amikor szükségünk lesz azokra.

```
# comand — formázóparancs végrehajtása
   subroutine comand(buf)
   character buf(MAXLINE)
   integer comtyp, getval, max
   integer argtyp, ct, spval, val
   include cpage
   include cparam

   ct = comtyp(buf)
   if (ct == UNKNOWN)          # ismeretlen parancsok kihagyása
      return
   val = getval(buf, argtyp)
   if (ct == FI) {
      call brk
      fill = YES
      }
   else if (ct == NF) {
      call brk
      fill = NO
      }
```

```
    else if (ct == BR)
        call brk
    else if (ct == LS)
        call set(lsval, val, argtyp, 1, 1, HUGE)
    # ...
    # stb.
    # ...
    return
    end
```

A legtöbb parancs egyszerűen csak egy paraméter új értékét állítja be, esetleg egy megszakítás kiváltása után. Megszakításkor a **brk** kerül hívásra a részben feltöltött sorok kivitele céljából. Pillanatnyilag megfelel a **brk** egy üres változata is, amely visszatér a hívóhoz anélkül, hogy bármit végrehajtana, mivel egyelőre kizárólag feltöltés nélküli szöveggel dolgozunk, így nincsenek részben feltöltött soraink.

A paraméterek többségét a **cparam** nevű közös adatmezőben tartjuk, mivel a programban jóformán mindenhol szükség van rájuk, ahhoz pedig túl sokan vannak, hogy szubrutin- vagy függvényparaméterként adjuk át őket. A többi paraméter a **cpage**-ban található, erre később visszatérünk.

```
common /cparam/ fill, lsval, inval, rmval, tival, ceval, ulval
    integer fill        # ha YES, feltöltés kell; kezdőért. = YES
    integer lsval       # aktuális sortávolság; kezdőért. = 1
    integer inval       # aktuális bekezdési mélység; > =0; kezdőért. = 0
    integer rmval       # aktuális sorvégződés; kezdőért. =
                        # PAGEWIDTH = 60
    integer tival       # aktuális ideiglenes bekezdés; kezdőért. = 0
    integer ceval       # középre állítandó sorok száma; kezdőért. = 0
    integer ulval       # aláhúzandó sorok száma; kezdőért. = 0
```

A **command** a parancsnév dekódolására **comtyp**-et, az esetleges paraméterek kiértékelésére pedig **getval**-t hívja. A **comtyp** annyi ismétlődést tartalmaz, hogy elegendő egy részét közölni:

```
# comtyp — parancs dekódolása
    integer function comtyp(buf)
    character buf(MAXLINE)

    if (buf(2) == LETF & buf(3) == LETI)
        comtyp = FI
    else if (buf(2) == LETN & buf(3) == LETF)
        comtyp = NF
    else if (buf(2) == LETB & buf(3) == LETR)
        comtyp = BR
    # ...
    # stb.
    # ...
    else
        comtyp = UNKNOWN
    return
    end
```

Ha a parancsok száma viszonylag kevés, az explicit vizsgálatokkal végzett direkt keresés a legkönnyebb és egyben megfelelő módszer. A végső változatban azonban esetleg kívánatos lehet a vizsgálatok helyett egy általánosabb, táblázatos keresési mód bevezetése.

Figyeljük meg, hogy a **comtyp** nem ellenőrzi, hogy a parancs pontosan két betűből áll-e, csupán azt vizsgálja, hogy az első két betű egyezik-e egy ismert parancs nevével. Ez lehetővé teszi, hogy a felhasználók — ha úgy tetszik — a parancsokat **.fill**, **.break** stb. alakban használják. A módszer hátránya, hogy minden újonnan bevezetett parancs első két betűjének különböznie kell a már meglevőkétől, ami csökkentheti a rövidítések mnemonikus értékét. Ennek alapján az Olvasó úgy is dönthet, hogy a **comtyp**-ben a teljes parancsnevet megvizsgálja.

Mivel majdnem minden parancs megengedi egy numerikus paraméter előjeles vagy előjel nélküli használatát, célszerű külön rutint írni a paraméter és az előjel átvételére. A **getval** átlépi a parancsot, az **argtyp**-ban feljegyzi, hogy van-e előjel, majd a numerikus paramétert a **ctoi** rutin segítségével egész számmá konvertálja.

```
# getval — esetleges numerikus paraméter kiértékelése
    integer function getval(buf, argtyp)
    character buf(MAXLINE)
    integer ctoi
    integer argtyp, i

    i = 1                          # a parancsnév átlépése
    while(buf(i) ¬= BLANK & buf(i) ¬= TAB & buf(i) ¬= NEWLINE)
        i = i + 1
    call skipbl(buf, i)            # a paraméter megkeresése
    argtyp = buf(i)
    if (argtyp == PLUS | argtyp == MINUS)
        i = i + 1
    getval = ctoi(buf, i)
    return
    end
```

Bár pillanatnyilag a parancsok kivétel nélkül egy pontból és két betűből állnak, a **getval** tetszőleges parancs átlépésére készült, amelyet szóköz, tabulátor vagy újsor határol. (Emlékezzünk vissza, hogy a **skipbl**, melyet a 6. fejezetben tárgyaltunk, átlépi a szóközöket és a tabulátorkaraktereket.) A parancs típusát pedig külön rutin, a **comtyp** dekódolja. Ezek a megoldások megkönnyítik majd a program módosítását, ha később erre szükség lenne. Kevés program marad változatlan élete folyamán; célszerű ezt már előzetesen is szem előtt tartani, hogy az elkerülhetetlen változtatások ne okozzanak nagy problémát.

A **getval**-t minden parancsnál behívjuk, még az olyan parancsok, mint a **.he** és **.fo** esetén is, melyekhez soha nem tartozhat numerikus paraméter. Ugyanis nem érdemes ezeket különlegesen kezelni; bár a formázó ilyenkor egy morzsányi fölösleges munkát végez, de a program szerkezete egyszerűbbé válik.

Egy paraméter új értékének beállítására a **set** általános rutint használjuk, akár abszolút, akár relatív, akár alapfeltételezés szerinti értékről legyen is szó. A **set** azt is megvizsgálja, hogy az eredményül kapott érték meghatározott intervallumba esik-e. Például, a **command**-ban a sortávolság beállítását elvégző programrész a következő:

```
    else if (ct == LS)
        call set(lsval, val, argtyp, 1, 1, HUGE)
```

A **set** hívásának eredménye az, hogy **lsval** értéke

val, ha a **.ls** parancsban nem volt előjel, mint pl. a **.ls 2**-ben;
lsval±**val**, ha volt előjel, mint pl. a **.ls —1**-ben;
és **lsval=1**, ha paraméter egyáltalán nem fordult elő.

Az eredményt minden esetben az utolsó két paraméter, 1 és **HUGE** (valamilyen nagy szám) közé kényszeríti.

```
# set — paraméterbeállítás és értékellenőrzés
    subroutine set(param, val, argtyp, defval, minval, maxval)
    integer max, min
    integer argtyp, defval, maxval, minval, param, val

    if (argtyp == NEWLINE)          # alapfeltételezés szerinti
        param = defval
    else if (argtyp == PLUS)        # relatív +
        param = param + val
    else if (argtyp == MINUS)       # relatív –
        param = param – val
    else                            # abszolút
        param = val
    param = min(param, maxval)
    param = max(param, minval)
    return
    end
```

Most már elkészítettünk annyit a parancselemzőből, hogy elvégezhessük a tesztelését, és így majd használni tudjuk és a program hátralevő részének kidolgozása folyamán már nem kell visszatérnünk rá.

Gyakorlat

7.3 A **format** figyelmen kívül hagyja az ismeretlen parancsokat. Úgy is kezelhetné őket, mint közönséges kiviendő szöveget, vagy mint hibákat, amelyeket jelezni kell. Vizsgáljuk meg mindkét lehetőség előnyeit! □

7.4 A parancsok egy, az első pozícióban álló ponttal kezdődnek, mivel ez szokványos szövegekben ritkán fordul elő. Milyen más, kézenfekvő lehetőség kínálkozik? Hogyan oldanánk meg egy ponttal kezdődő sor kinyomtatását? Tegyük fel, hogy parancsokat és szöveget vegyesen akarunk használni egy soron belül. Jó ötlet ez? Melyik az a szintaxis, amelyet könnyű leírni és szerkeszteni? □

7.4 A LAPOK FORMÁTUMA

A következő lépés a szöveg kezelésének ésszerű lépésekre való felbontása. Tekintet nélkül arra, hogy szükség van-e feltöltésre, középre állításra vagy a sorok aláhúzására, a **format** minden lapra a megfelelő számú sort kell, hogy nyomtassa. Foglalkozzunk hát a szövegsoroknak a bemenetről a kimenetre való átmásolásával, betartva a megfelelő sortávolságot, a fej- és lábrészt, valamint a lapszámozást. Ezzel

majdnem teljesen kimerítettük a feltöltés nélküli üzemmódot. A program hasonlít a 3. fejezetben szereplő **print**-hez, ezért hasznosíthatunk néhány ott szerzett tapasztalatot. A legfontosabb a határesetek jelentősége — minden lapra megfelelő számú sor kerüljön a megfelelő helyre, felesleges lapokat pedig ne hozzunk létre.

Egy lap függőleges felosztását leíró paraméterek a következők: **plval** a lap hossza; a lap tetején levő üres részek a fejléc előtt és után: **m1val** és **m2val** (**m1val**-ba beleértendő a fejléc is); a megfelelő lap alján levő üres részek: **m3val** és **m4val**; és végül az utolsó sor, ahová még szöveg kerülhet: a **bottom.** Fennáll a következő összefüggés:

```
bottom = plval – m3val – m4val
```

A 11 hüvelykes (kb. 28 cm) papírméret és a szabványos 6 sor/hüvelyk sortávolság esetén **plval**=66. Ha a fej és a láb szövege körüli üres részek mindegyike 2—2 sor, a lapra 58 szövegsor kerül, és **bottom**=62.

A kinyomtatandó lap következő sorának sorszáma **lineno**; ha értéke zérus, a lap tetején vagyunk, ha nagyobb mint a **bottom**, akkor elértük a lap végét. Amikor **lineno** átlépi a **bottom** értékét, akkor az aktuális oldalt ki kell nyomtatni. Az aktuális oldal sorszáma a **curpag**; a következő lapra kerülő sorszám pedig a **newpag.** Ezeket az értékeket a **cpage** nevű közös adatmezőben tartjuk, ahol az ismétlődő fej- és lábszöveg is helyet kap.

```
common /cpage/ curpag, newpag, lineno, plval, m1val, m2val, m3val,
        m4val, bottom, header(MAXLINE), footer(MAXLINE)
    integer curpag      # aktuális lapsorszám; kezdőért. = 0
    integer newpag      # következő lapsorszám; kezdőért.=1
    integer lineno      # következőként nyomtatandó sor; kezdőért.=0
    integer plval       # lapméret sorokban; kezdőért.=PAGELEN=66
    integer m1val       # fejléc előtti üres sorok száma a fejjel együtt
    integer m2val       # fejléc utáni üres sorok száma
    integer m3val       # utolsó szövegsor utáni üres sorok száma
    integer m4val       # lap alján levő üres sorok száma a lábbal együtt
    integer bottom      # a lap utolsó értékes sora=plval–m3val–m4val
    character header    # a fej szövege; kezdőért.=NEWLINE
    character footer    # a láb szövege; kezdőért.=NEWLINE
```

Most már nem egyszerűen átmásolni akarjuk a szövegsorokat a standard outputra, ezért a **text**-et kissé módosítjuk, hogy egy új **put** rutint hívjon a **putlin** helyett:

```
# text — szövegsorok feldolgozása (első, nyers változat)
    subroutine text(inbuf)
    character inbuf(INSIZE)

    call put(inbuf)
    return
    end
```

A fej és a láb szövegétől, valamint a **.sp** által generált üres soroktól eltekintve, minden kimenő szövegsort a **put** vezérel. A **put** és alárendelt rutinjai gondoskodnak a lap tetején és alján kihagyandó sorokról, a sortávolságról, a lapszámozásról és a bekezdésekről, melyekre rövidesen kitérünk. A **put** vázlata a következő:

```
if (lap tetején vagy lap alján túl vagyunk)
    állíts elő fejlécet a hozzá tartozó üres sorokkal
    vidd ki a bekezdési mélységhez szükséges szóközöket, ha szükséges
    vidd ki a sort
    növeld a sor sorszámát
if (túl vagyunk a lap alján)
    állíts elő láblécet a hozzá tartozó üres sorokkal
```

A tényleges program pedig:

```
# put — sor kivitele megfelelő sortávolsággal és bekezdési mélységgel
    subroutine put(buf)
    character buf(MAXLINE)
    integer min
    integer i
    include cpage
    include cparam

    if (lineno == 0 | lineno > bottom)
        call phead
    for (i = 1; i <= tival; i = i + 1)      # bekezdés
        call putc(BLANK)
    tival = inval
    call putlin(buf, STDOUT)
    call skip(min(lsval-1, bottom-lineno))
    lineno = lineno + lsval
    if (lineno > bottom)
        call pfoot
    return
    end
```

A **skip(n)** *n* számú üres sort generál (amelyek egyetlen **NEWLINE**-ból állnak), ha *n* pozitív, és hatástalan, ha *n* kisebb, mint 1. A rutint a **print**-hez írtuk, lásd a 3. fejezetet. A bekezdést kezelő programrész szintén beépített a **put**-ba; feladata csupán annyi, hogy kiviszi a megfelelő számú soreleji szóközt és visszaállítja az esetleges ideiglenes bekezdési mélységet, ezért pillanatnyilag figyelmen kívül hagyhatjuk.

A **put**-nak értelmesen kell viselkednie akkor is, ha szokatlan paramétereket kap. Így például előfordulhat, hogy **lsval** sortávolság értéke nagyobb, mint a lábnál kihagyandó sorok száma. Ezért van arra szükség, hogy egy sor kivitele után a következő **skip** hívás legfeljebb csak a **bottom+1** sorszámú sorig juthasson el. (A **put** csak **lsval+1** számú üres sort generál, mivel **putlin** már létrehozott egy sort.) Mivel minden oldal a lap tetején kezdődik, a sortávolságtól függetlenül minden lapra kerül legalább egy sor, és biztosítjuk azt is, hogy a **format** valamilyen mértékben előrehaladjon a szövegben, bármilyen szokatlan paramétereket is kap.

A **phead** és a **pfoot** írja ki a fej- és a láblécet. A **phead** végzi el az aktuális és az új lapszám módosítását **curpag**-ban, illetve **newpag**-ban.

```
# phead — fej kivitele
    subroutine phead
    include cpage
```

```
    curpag = newpag
    newpag = newpag + 1
    if (m1val > 0) {
        call skip(m1val-1)
        call puttl(header, curpag)
        }
    call skip(m2val)
    lineno = m1val + m2val + 1
    return
    end

# pfoot — láb kivitele
    subroutine pfoot
    include cpage

    call skip(m3val)
    if (m4val > 0) {
        call puttl(footer, curpag)
        call skip(m4val-1)
        }
    return
    end
```

A fej **m1val** utolsó, a láb pedig **m4val** első sora. Ennek megfelelően akár a fej, akár a láb, akár a lapokra való tördelés megszüntethető, ha **m1val, m2val** vagy mindkettő értéke zérus. Ez részletkérdés, de megold egy különben kivételnek számító esetet.

A **phead** és a **pfoot** a fej, illetve a láb különálló sorként való kiírására a **puttl**-t hívja, amely, ha ez szükséges, a lapszámot a **putdec** segítségével állítja elő.

```
# puttl — fej vagy láb szövegsorának kivitele egy esetleges lapszámmal
    subroutine puttl(buf, pageno)
    character buf(MAXLINE)
    integer pageno
    integer i

    for (i = 1; buf(i) ¬= EOS; i = i + 1)
        if (buf(i) ==PAGENUM)
            call putdec(pageno, 1)
        else
            call putc(buf(i))
    return
    end
```

A **PAGENUM** azt a karaktert jelöli, amelyet a fej vagy lábban a lapszámmal kell helyettesíteni. Mi a # számjelet használjuk, amelynek talán bizonyos mnemonikus értéke is van. A programban azért fordul elő külön néven, hogy szerepe világos, módosítása pedig egyszerűbb legyen.

A fej és a láb szövegét eredetileg a **gettl** veszi elő, melyet a **command** hív a következő formában:

```
else if (ct == HE)
    call gettl(buf, header)
    else if (ct == FO)
    call gettl(buf, footer)
```

A **gettl** maga pedig:

```
# gettl — fej vagy láb másolása buf-ból ttl-be
    subroutine gettl(buf, ttl)
    character buf(MAXLINE), ttl(MAXLINE)
    integer i

    i = 1                      # a parancsnév átlépése
    while(buf(i) ¬= BLANK & buf(i) ¬= TAB & buf(i) ¬= NEWLINE)
        i = i + 1
    call skipbl(buf, i)  # a paraméter keresése
    if (buf(i) == SQUOTE | buf(i) == DQUOTE)
                               # az idézőjel elhagyása, ha van ilyen
        i = i + 1
    call scopy(buf, i, ttl, 1)
    return
    end
```

A fej- vagy láblécről feltesszük, hogy a legközelebbi szóköztől különböző karakterrel kezdődik, de a kezdetén álló egyszeres vagy kettős idézőjelet figyelmen kívül hagyjuk lehetővé téve ezáltal azt is, hogy a fej- vagy lábléc szóközökkel kezdődhessék.

Figyeljük meg, hogy a **gettl** hívását a

```
call gettl(buf, header)
```

illetve a

```
call gettl(buf, footer)
```

utasításokkal végezzük, ahelyett, hogy a parancs típusát adnánk át a **gettl**-nek és rábíznánk annak eldöntését, hogy fej-, vagy láblécet kell generálnia. Ezen utóbbi megoldás ugyanis megköveteli, hogy **gettl** ismerje a **cpage**-beli változók formáját, ami a program adatkapcsolatainak számát növelné. Ha van rá mód, rejtsük el a részleteket azon rutinok elől, amelyeknek nincs ezekre szükségük.

A **space**-t közvetlenül a **command** hívja, valahányszor egy **.sp**-vel találkozik. Ha paraméter nincs, egyetlen üres sor jön létre. A **.sp 0** teljesen szabályos, egyetlen hatása egy megszakítás.

```
else if (ct == SP) {
    call set(spval, val, argtyp, 1, 0, HUGE)
    call space(spval)
    }
```

A **space**-t hívjuk a **.bp** parancs esetén is, hogy elérjünk az aktuális oldal aljára.

```
else if (ct == BP) {
  if (lineno > 0)
       call space(HUGE)
call set(curpag, val, argtyp, curpag + 1, -HUGE, HUGE)
newpag = curpag
}
```

Ha nem fordítunk különleges gondot a **.bp** programjára, kellemetlen meglepetésben lehet részünk. Először is, a **.bp** és a **.bp +1** parancsok hatása a specifikáció értelmében azonos kell, hogy legyen. A **.bp** parancs a lap alján (még az utolsó lap esetén is) hatástalan, a normális lapszámnöveléstől eltekintve. Ugyancsak meg kell győződnünk arról, hogy egy szöveget akár a **.bp**, **.bp +1**, **bp1** parancsokkal, akár ezek nélkül kezdve, az első lap sorszáma 1 lesz. Végül, a **.bp** parancs negatív lapszámokat is megenged. Nem azért, mert azt gondoltuk, hogy valaki valaha is használja ezeket, hanem azért, mert semmi előre látható kárt nem okoznak. Az önkényes megszorítások egyszer zavarhatnak valakit.

Ha egyszer a **put** már működik, a **space** hasonlóan írható meg. A **space(n)** hatástalan, ha a lap alján fordul elő (**lineno > bottom**). Különben *n* számú, de legfeljebb annyi üres sort állít elő, amennyi a lap aljáig hátravan. Ha elér a lap aljára, a lapot befejező üres sorok is létrejönnek.

```
# space — n üres sor kivitele vagy megállás a lap alján
   subroutine space(n)
   integer min
   integer n
   include cpage

   call brk
   if (lineno > bottom)
        return
   if (lineno == 0)
        call phead
   call skip(min(n, bottom+1-lineno))
   lineno = lineno + n
   if (lineno > bottom)
        call pfoot
   return
   end
```

Fontos, hogy a „lap alját" megkülönböztessük a „következő lap tetejétől", mivel ez a két hely nem ugyanaz. Ha egy **.sp** egy lap tetejénél fordul elő, az üres sorok létrejönnek, de ilyen eset általában csak a teljes szöveg elején fordul elő. Egy lap alja utánra jutó üres sorok elvesznek, mivel a felhasználó általában ezt várja. Ha egy lap tetején ténylegesen üres sorokra van szükség, ezek — mint erre rövidesen rátérünk —, csupa szóközből álló szövegsorokkal állíthatók elő.

A **space** és a **put** nagyon hasonlóak, ezért módosíthatnánk úgy a **space**-t, hogy az **put**-ot hívja. Ily módon — helyesen — egy helyre összpontosítanánk a kimenetet. A fő probléma ezzel az, hogy hogyan tudjuk elkerülni **lsval** számú üres sor kivitelét, valahányszor a javasolt új **space** behívja **put**-ot. Semmiképpen sem kívánatos **lsval** értékét módosítani a **space**-ban a **put** hívása előtt, majd utána vissza-

állítani. Ez a kétbalkezes adatkezelés műfajába tartozó legrosszabb lépés, mivel mindennek inkább mondható, mint kézenfekvőnek. Nem akarjuk **put** minden hívását egy sortávolság-paraméterrel terhelni, még kevésbé akarjuk, hogy egy bekezdéssel kivitt szöveg esetén a **space** által produkált üres sorok is soreleji szóközöket tartalmazzanak. Egy programrész látszólagos ismétlésének ellenére az eredeti megoldást célszerű választani.

Most már elég nyers programunk van a sortávolság, az üres sorok, az új oldalak stb. minden változatának kipróbálásához. A **.pl** parancs a tesztelés során különösen hasznos, mivel segítségével olyan méretűre rövidíthetjük az oldalt, hogy a kivitt sorokat kézi úton is könnyen megszámolhatjuk.
A **command**-ban **.pl** programja a következő:

```
else if (ct == PL) {
     call set(plval, val, argtyp, PAGELEN,
          m1val+m2val+m3val+m4val+1, HUGE)
     bottom = plval — m3val — m4val
     }
```

Célszerű lehet a **skip**-et úgy módosítani, hogy minden kivitt sorba elhelyezzen egy látható karaktert; ez megkönnyíti a kinyomtatott sor értelmezését.

Melyek azok a különösen kellemetlen határesetek, amelyek bajt okozhatnak? Legalább egy hiba van — az utolsó kivitt oldal rövid, mivel nem gondoskodtunk azon többletsorok kiviteléről, melyekkel elértünk volna ezen oldal alját. A hibát úgy javítjuk, hogy nagyszámú üres sor kivitelét kérjük a főprogramban is input végének elérése után, akárcsak a **.bp** parancs esetében. A **format** átdolgozott változata tehát:

```
# format — szövegformázó főprogram (végleges változat)
    character inbuf(INSIZE)
    integer getlin
    include cpage

    call init
    while (getlin(inbuf, STDIN) ¬= EOF)
         if (inbuf(1) == COMMAND)       # parancs
              call comand(inbuf)
         else                           # szöveg
              call text(inbuf)
    if (lineno > 0)
         call space(HUGE)               # maradék output kivitele
    stop
    end
```

Ez jól működik (azaz nem csinál semmit), ha már az utolsó lap alján vagyunk, mivel ott további üres sor nem keletkezik. Jól működik abban a kevésbé megszokott esetben is, amikor egy lap tetején vagyunk; pontosabban, ha nincs bemenet, nem ad kimenetet sem.

Írhattunk volna egy külön programrészt is az utolsó lap végének kezelésére, de sokkal célszerűbb, amikor csak lehet, a fentihez hasonló, már meglevő szerkezeteket felhasználni. Ezzel elkerüljük azt, hogy ugyanazt a dolgot két, egymástól kissé különböző módon oldjuk meg (melyek közül az egyikről elfelejtkezünk majd, amikor

a másikat tökéletesítjük); és rákényszerítjük magunkat arra, hogy a mindenütt használt eljárást a fontos határesetekre vonatkozóan is alaposan átgondoljuk.

Ezzel a **format** alapszerkezete lényegében kialakult, ezért bemutathatjuk a hierarchia jelenlegi állapotát:

```
format
    init
    getlin
    comand
        comtyp, getval, brk, set, gettl, space
    text
        put
            phead (pfoot)
                skip
                    putc
                puttl
                    putc, putdec
            putc
            putlin
            skip
        space
            brk, phead, pfoot, skip
    space
```

A **phead** és a **pfoot** ugyanazokat a rutinokat hívja, ezért közösen szerepelnek. A további szubrutinok legtöbbjét a **text**-ből fogjuk hívni.

Gyakorlat

7.5 Valóban szükség van a **curpag** és a **newpag** nevű változókra az oldalszámok nyilvántartásához? Próbáljuk meg átírni a programot úgy, hogy csak az egyiket használjuk. Győződjünk meg róla, hogy a **.bp** és **.bp +1** ekvivalens maradt. □

7.5 BEKEZDÉSEK

A következő lépés az állítható sorkezdet és sorvégződés megvalósítása, mivel ezekre szükség van a feltöltött szöveg kezeléséhez. Minden kiviendő sorhoz ismernünk kell az **inval** bekezdési mélységet, az **rmval** sorvégződést és az ideiglenes bekezdési mélységet, **tival**-t. A **tival** mutatja a kinyomtatandó sort megelőző szóközök számát. A **.in**, **.rm** és **.ti** parancsoknak megfelelő **command**-beli programrész a következő:

```
else if (ct == IN) {
    call set(inval, val, argtyp, 0, 0, rmval-1)
    tival = inval
    }
else if (ct == RM)
    call set(rmval, val, argtyp, PAGEWIDTH, tival+1, HUGE)
```

```
else if (ct == TI) {
    call brk
    call set(tival, val, argtyp, 0, 0, rmval)
    }
```

A **put** által előállított sorokat **tival** számú szóköz előzi meg. Minden kivitt sor után a **tival**-t vissza kell állítani **inval** értékére, hogy az ideiglenes bekezdési mélység csak egyetlen sort érintsen. Ezt a **put**-on belül a következő programrész végzi el:

```
for (i = 1; i <= tival; i = i + 1)      # bekezdési mélység
    call putc(BLANK),
tival = inval
```

A sor eleji szóközöket és az üres sorokat mint speciális értékeket a **text**-ben a következő utasítások szűrik ki:

```
if (inbuf(1) == BLANK | inbuf(1) == NEWLINE)
    call leadbl(inbuf)       # mozgatás balra, tival állítása
```

Ezek az utasítások a **put** hívása előtt hajtódnak végre. A sor eleji szóközöket a **leadbl** kezeli:

```
# leadbl — vezető szóközök törlése, tival állítása
    subroutine leadbl(buf)
    character buf(MAXLINE)
    integer max
    integer i, j
    include cparam

    call brk
    for (i = 1; buf(i) == BLANK; i = i + 1)  # az első nem szóköz
                                             # keresése
        ;
    if (buf(i) ¬= NEWLINE)
        tival = i - 1
    for (j = 1; buf(i) ¬= EOS; j = j + 1) {  # a sor balra mozgatása
        buf(j) = buf(i)
        i = i + 1
        }
    buf(j) = EOS
    return
    end
```

A **leadbl** a teljes sort balra mozgatja el, úgy hogy az első nem szóköz-karakter az első pozícióba kerüljön. Ez biztosítja, hogy a sor eleji szóközök feltöltés nélküli üzemmódban nem kapnak kétszeres bekezdési mélységet, egyszer az ideiglenes bekezdés miatt, egyszer pedig a vezető szóközök következtében.

7.6 FELTÖLTÖTT SZÖVEG

Most, hogy a feltöltés nélküli szöveggel és a lapszélekkel rendben vagyunk, rátérünk a feltöltött szövegre, amit a felhasználók leggyakrabban igényelnek. Hogyan működik ez? Két rutin, a **getwrd** és a **putwrd** korutinszerűen működik együtt. A **getwrd** a bemenetet szavakra tagolja, és átadja **putwrd**-nek. A **putwrd** újra összerakja a szöveget és időnként kiad egy feltöltött sort. Így a **text** a következőképpen alakult:

```
# text — szövegsorok feldolgozása (második ideiglenes változat)
   subroutine text(inbuf)
   character inbuf(INSIZE), wrdbuf(INSIZE)
   integer getwrd
   integer i
   include cparam

   if (inbuf(1) == BLANK | inbuf(1) == NEWLINE)
      call leadbl(inbuf)            # mozgatás balra, tival állítása
   if (inbuf(1) == NEWLINE)         # üres sor
      call put(inbuf)
   else if (fill == NO)             # feltöltés nélküli szöveg
      call put(inbuf)
   else                             # feltöltött szöveg
      for (i = 1; getwrd(inbuf, i, wrdbuf) > 0;)
         call putwrd(wrdbuf)
   return
   end
```

A **getwrd**-ot a 3. fejezetben szereplő **include** program hatására írtuk. Feladata, hogy szétválassza a szavakat, azaz a szóköztől különböző karaktersorozatokat. Behíváskor paraméterként az **inbuf** indexét kapja, amely megmutatja, hogy honnan kezdve keresse a szót. Kilépéskor ezt az indexet úgy állítja át, hogy a megtalált szót követő pozícióra mutasson, így készen áll a következő hívásra. A sor végén **getwrd** nullát ad vissza, sor közben pedig a talált szó pozitív hosszát.

A folyamat másik végén a **putwrd** áll. Ha az új szó nem fér el az aktuális sorban, **putwrd** a **brk** behívásával kiadja a sort és felkészül az újabb sorra. Különben az új szó a sor végére kerül. A szó után **putwrd** elhelyez egy szóközt is, hogy az a következő szótól megfelelően el legyen választva.

A főprogramban az **EOF** utáni **space** hívás most már kettős célt szolgál: mivel megszakítást hoz létre, kikényszeríti a **putwrd** által felépített esetleges töredéksor kivitelét, mielőtt üres sorokkal elmenne az utolsó lap aljáig. Ez újabb előnye annak, hogy a magasszintű **space** függvényt használtuk egy „utolsó oldal" rutin megírása helyett.

```
# putwrd — egy szó elhelyezése outbuf-ban
   subroutine putwrd(wrdbuf)
   character wrdbuf(INSIZE)
   integer length, width
   integer last, llval, w
   include cout
   include cparam
```

```
    w = width(wrdbuf)
    last = length(wrdbuf) + outp + 1    # outbuf új vége
    llval = rmval - tival
    if (outp > 0 & (outw+w > llval | last >= MAXOUT)) { # túl nagy
        last = last - outp              # wrdbuf végének megjegyzése
        call brk                        # előző sor kivitele
        }
    call scopy(wrdbuf, 1, outbuf, outp+1)
    outp = last
    outbuf(outp) = BLANK        # szóköz a szavak között
    outw = outw + w + 1         # 1 a szóköz miatt
    outwds = outwds + 1
    return
    end
```

A kivitelre kerülő sor **outbuf**-ban gyűlik össze. Az utolsó karakterpozíciót **outp**, a sor szélességét pedig **outw** jelöli. A szavakat **outwds**-ben számláljuk. Ezen változókat a **cout** közös adatmezőben tartjuk, melyet **brk** is használ.

```
common /cout/ outp, outw, outwds, outbuf(MAXOUT)
    integer outp          # outbuf utolsó kar. pozíciója; kezdőért.=0
    integer outw          # szöveg szélessége outbuf-ban; kezdőért.=0
    integer outwds        # szavak száma outbuf-ban; kezdőért.=0
    character outbuf      # a feltöltendő sorokat gyűjtjük itt
```

Az aktuális sor szélessége, **outw,** nem ugyanaz, mint **outp,** ami a sor utolsó karakterére mutat. Miért? Ha ugyanis egy felhasználó történetesen olyan sort ír, amely visszalépés-karaktert tartalmaz pl. egy karakter aláhúzásának előkészítésére, akkor nyilvánvalóan, az

```
    a BACKSPACE b
```

sorozat hosszának minden ésszerű definíció alapján egynek kell lennie, hiszen a kimeneten egy pozíciót foglal el; ugyanakkor a sor „hossza" — a tényleges tárigény szerint — három (karakter). A két mérték tehát különböző. (Az aláhúzóparancsot még nem dolgoztuk ki.)

Ezen probléma megoldására (valamint az aláhúzási művelet előkészítésére) a „szélességet" és a „karakterek számát" megkülönböztettük, és a szélességszámítást egy külön függvény végzi el.

```
# width — karaktersorozat szélességének meghatározása
    integer function width(buf)
    character buf(MAXLINE)
    integer i
```

```
width = 0
for (i = 1; buf(i) ¬= EOS; i = i + 1)
    if (buf(i) == BACKSPACE)
        width = width - 1
    else if (buf(i) ¬= NEWLINE)
        width = width + 1
return
end
```

A **width**-ben két említésre méltó szimbolikus konstans van: a **BACKSPACE** melynek szélessége −1 és a **NEWLINE**, amelyé zérus. Minden más karakter szélessége 1.

Most már abban a helyzetben vagyunk, hogy bemutathatjuk a **brk** rutint. Tulajdonképpen nem kell sokat mondani róla: a **put** felhasználásával kiviszi az **outbuf**-ban található szöveget, nullára állítja a kivitelhez tartozó mutatót, valamint a szélesség- és a szószámlálót.

```
# brk — az aktuális feltöltött sor vége
    subroutine brk
    include cout

    if (outp > 0) {
        outbuf(outp) = NEWLINE
        outbuf(outp + 1) = EOS
        call put(outbuf)
        }
    outp = 0
    outw = 0
    outwds = 0
    return
    end
```

Gyakorlatok

7.6 A visszalépésen és az új soron kívül mely karakterek szélessége zérus az Olvasó által használt rendszerben? Elképzelhetők-e olyan karakterek, melyek szélessége nem állandó? □

7.7 Ha az Olvasó által használt jelkészlet nem tartalmaz visszalépés-karaktert, hogy lehetne ezt megvalósítani a **format** felhasználóinak számára? □

7.7 A SORVÉGZŐDÉSEK KIEGYENLÍTÉSE

Most már nem maradt más hátra, mint a kivitelre kerülő sorok végződéseinek egymáshoz igazítása, vagyis az egyenes laptükör előállítása. A megfelelő hely ennek elvégzésére a **putwrd**, közvetlenül a **brk** hívása előtt, amikor is egy teljes sorral rendelkezünk, tudjuk, hogy feltöltött szöveggel dolgozunk, ismerjük a sorban levő szavak számát, és ezek szélességét. A **putwrd**-n most olyan módosítást hajtunk végre, hogy hívjon be egy külön rutint, amely széttolja a sort és külön szóközöket iktat be.

```
# putwrd — egy szó behelyezése outbuf-ba, sorvég-kiegyenlítéssel
   subroutine putwrd(wrdbuf)
   character wrdbuf(INSIZE)
   integer length, width
   integer last, llval, nextra, w
   include cout
   include cparam

   w = width(wrdbuf)
   last = length(wrdbuf) + outp + 1         # outbuf új vége
   llval = rmval - tival
   if (outp > 0 & (outw+w > llval | last >= MAXOUT)) { # túl nagy
      last = last - outp          # wrdbuf végének megjegyzése
      nextra = llval - outw + 1
      call spread(outbuf, outp, nextra, outwds)
      if (nextra > 0 & outwds > 1)
         outp = outp + nextra
      call brk                    # előző sor kivitele
      }
   call scopy(wrdbuf, 1, outbuf, outp+1)
   outp = last
   outbuf(outp) = BLANK         # szóköz a szavak között
   outw = outw + w + 1          # 1 a szóköz miatt
   outwds = outwds + 1
   return
   end
```

A **spread** a soron belül, a jobb oldali első szóval kezdve, a szavakat jobbra mozgatja. Valahányszor egy szó elmozdul, a sor néhány szóközzel bővül, melyek amennyire lehet, egyenletesen oszlanak el. A folyamat befejeződik, amikor több szóköz beiktatására már nincs szükség. A **nextra** változó tartalmazza a kiegyenlítéshez szükséges többletszóközök számát. Ügyelnünk kell arra a speciális esetre, amikor egy hosszú bemenő szó **nextra**-t negatívvá teszi, valamint arra, amikor a sorban csak egyetlen szó áll.

Olyan esetekben, amikor a többletszóközök nem oszthatók el egyenletesen a sorban, esztétikai meggondolásból a fennmaradókat soronként váltakozva jobb és bal oldalról indulva osztjuk szét, hogy ezáltal elkerüljük a szóközökből álló folytonos függőleges vonalak („fehér folyamok") kialakulását valamelyik lapszélen. A **dir** értéke váltakozva 0 és 1; ez jelöli ki, hogy a fennmaradó szóközök szétosztása melyik oldalról induljon.

```
# spread — szavak széttolása, sorvég-kiegyenlítés
   subroutine spread(buf, outp, nextra, outwds)
   character buf(MAXOUT)
   integer min
   integer dir, i, j, nb, ne, nextra, nholes, outp, outwds
   data dir /0/
```

```
if (nextra <= 0 | outwds <= 1)
    return
dir = 1 - dir              # az előző irány megfordítása
ne = nextra
nholes = outwds - 1
i = outp - 1
j = min(MAXOUT-2, i+ne)           # hely a NEWLINE és EOS
                                  # számára
while (i < j) {
    buf(j) = buf(i)
    if (buf(i) == BLANK) {
        if (dir == 0)
            nb = (ne-1) / nholes + 1
        else
            nb = ne / nholes
        ne = ne - nb
        nholes = nholes - 1
        for (; nb > 0; nb = nb - 1) {
            j = j - 1
            buf(j) = BLANK
            }
        }
    i = i - 1
    j = j - 1
    }
return
end
```

Ez némileg ravasz program (ami nem válik éppen dicséretére), de helyesen old meg egy bonyolult feladatot. A csel az **nb** kiszámításában van, amely a többletszóközöket a lehető legegyenletesebben osztja ki, a fennmaradókat pedig váltakozva a sor más-más végéről indulva helyezi el. A nullával való osztás veszélye nem áll fenn, jóllehet az **nholes** értéke állandóan csökken, ugyanis a program csak akkor hajtódik végre, ha **nextra > 0** és **outwds > 1**, a ciklusból pedig kilépünk, ha **nholes** értéke 1.

Mellesleg, érdemes megfigyelni, hogy a **spread**-ben megvizsgáltuk az

```
if (nextra <= 0 | outwds <= 1)
```

feltételt, majd megismételtük az ezzel azonos vizsgálatot

```
if (nextra > 0 & outwds > 1)
```

formában a **putwrd**-ben.
Miért nem írtuk ehelyett **putwrd**-ben a következőt:

```
if (nextra > 0 & outwds > 1) {
    call spread(outbuf, outp, nextra, outwds)
    outp = outp + nextra
    }
```

majd miért nem hagytuk el **spread**-ből a felesleges vizsgálatot?

Ennek oka az volt, hogy ha így járunk el, a hívó programnak ismernie kellene a hívott program paramétereire vonatkozó megszorításokat. Ez a rejtett kapcsolatok, illetve a függőség egy újabb formája. A program hosszú távú karbantartását tekintve veszélyeket rejt magában, mivel előbb vagy utóbb valaki a hívó program módosításakor megsérti a megszorításokat és ezzel esetleg hibákat idéz elő. Ezért úgy építettük fel a **spread**-et, hogy ellenőrizze saját paramétereit ahelyett, hogy a **putwrd**-ben bízna. Természetesen az ilyen jellegű dolgokat túlzásba is lehet vinni, de ennek költsége itt jelentéktelen, ezért érdemes vele foglalkozni.

Gyakorlatok

7.8 Mutassuk be, hogy a **spread** jól működik, akár a jobb, akár a bal oldalon helyezi is el a fennmaradó szóközöket. Bizonyítsuk be, hogy ésszerűen viselkedik még **outwds, nextra** és **outp** szélső értékeinél is. Hogyan lehetne a **format**-ot másképpen szervezni, hogy a **spread** egyszerűbb legyen? □

7.9 Egészítsük ki a parancsok készletét a **.ju** *(justify)* sorigazítási, és a **.nj** *(no justify)* sorigazítást tiltó parancsokkal úgy, hogy a sorigazítás a feltöltéstől függetlenül ki és bekapcsolható legyen. Célszerű-e feltöltetlen sorok igazítását megengedni? □

7.8 KÖZÉPRE ÁLLÍTÁS ÉS ALÁHÚZÁS

Miután a formázó nagy része működik, hozzáfoghatunk a feldíszítéséhez.

Az egyik leghasznosabb bővítés a középre állítás, mivel ez egy olyan aprólékos, könnyen elrontható műveletet gépesít, melyet embernek nem is lenne szabad végeznie. Megvalósítása szerencsére igen könnyű. Egy **.ce** parancs előfordulásakor, a **command** meghatározza a sorok számát, és elhelyezi **ceval**-ban a következő utasítások segítségével:

```
else if (ct == CE) {
     call brk
     call set(ceval, val, argtyp, 1, 0, HUGE)
     }
```

Valahányszor egy középre állított sor kinyomtatódik, a **text** eggyel csökkenti **ceval** értékét; amikor **ceval** eléri a nullát, további középre állításra nincs szükség. Ez a programrész a **text**-be kerül:

```
if (ceval > 0) {          # középre állítás
     call center(inbuf)
     call put(inbuf)
     ceval = ceval - 1
     }
```

Végül a sort kiírás előtt középre kell állítani. Ez egy alkalmas ideiglenes bekezdési mélység beállításával történik, amely a sort a megfelelő mértékben jobbra tolja; kivitelkor a sor már a kívánt pozícióban helyezkedik el. Ezt a tevékenységet külön rutinként írtuk meg annak ellenére, hogy mindössze egyetlen sorból áll; úgy találtuk azonban, hogy így kevésbé teszi a **text**-et zavarossá. Az Olvasó más megoldás mellett is találhat érveket.

```
# center — egy sor középre állítása tival segítségével
    subroutine center(buf)
    character buf(ARB)
    integer max, width
    include cparam

    tival = max((rmval+tival-width(buf))/2, 0)
    return
    end
```

Az aláhúzást szintén nehézkes kézzel végezni; a középre állításhoz hasonlóan ezt is célszerű gépesíteni. Az aláhúzáshoz szükséges tennivalók sorrendje lényegében azonos a középre állításéval. A **command** beállítja **ulval**-t:

```
else if (ct == UL)
    call set(ulval, val, argtyp, 0, 1, HUGE)
```

A **text** csökkenti **ulval**-t minden aláhúzott bemenő sornál:

```
if (ulval > 0) {            # aláhúzás
    call underl (inbuf, wrdbuf, INSIZE)
    ulval = ulval - 1
    }
```

Végül, az **underl** nevű külön rutin készíti elő a szavak aláhúzását oly módon, hogy minden — szóköztől, tabulátortól vagy visszalépéstől különböző — karaktert

karakter **BACKSPACE UNDERLINE**

formára alakít át. A **text** ezután középre állítja a sort és feltöltés nélkül kiviszi vagy szavanként feldolgozza a feltöltött szöveg számára.

```
# underl — egy sor aláhúzása
    subroutine underl(buf, tbuf, size)
    integer i, j, size
    character buf(size), tbuf(size)

    j = 1          # kibővítés tbuf-ban
    for (i = 1; buf(i) ¬= NEWLINE & j < size-1; i = i+1){
        tbuf(j) = buf (i)
        j = j + 1
        if (buf(i) ¬= BLANK & buf(i) ¬= TAB
                & buf(i) ¬=BACKSPACE) {
            tbuf(j) = BACKSPACE
            tbuf(j+1) = UNDERLINE
            j = j + 2
            }
        }
    tbuf(j) = NEWLINE
    tbuf(j + 1) = EOS
    call scopy(tbuf, 1, buf, 1)      # visszamásolás buf-ba
    return
    end
```

A visszalépések és az aláhúzások úgy kerülnek a szövegbe, hogy azt átmásoljuk egy ideiglenes tömbbe, majd visszavisszük eredeti helyére. A művelet helyben is elvégezhető lenne, de ez a programot jelentősen elbonyolítaná.

Lényeges, hogy ezeket a tevékenységeket a **text**-en belül a megfelelő sorrendben hajtsuk végre. Az aláhúzást és a sor eleji szóközök vizsgálatát kell először elvégezni, mivel a többi eset kikényszeríti a kivitelt. A középre állítás meg kell, hogy előzze a **NEWLINE** vizsgálatot, így egy középre állított üres sor is csökkenti **ceval**-t. A **text** végső változatát a fenti tevékenységek összegyűjtésével kapjuk:

```
# text — szövegsorok feldolgozása, végleges változat
    subroutine text(inbuf)
    character inbuf(INSIZE), wrdbuf(INSIZE)
    integer getwrd
    integer i
    include cparam

    if (inbuf(1) == BLANK | inbuf(1) == NEWLINE)
        call leadbl(inbuf)   # mozgatás balra, tival állítása
    if (ulval > 0) {         # aláhúzás
        call underl(inbuf, wrdbuf, INSIZE)
        ulval = ulval - 1
        }
    if (ceval > 0) {         # középre állítás
        call center(inbuf)
        call put(inbuf)
        ceval  = ceval - 1
        }
    else if (inbuf(1) == NEWLINE)        # üres sor
        call put(inbuf)
    else if (fill == NO)     # feltöltetlen szöveg
        call put(inbuf)
    else                     # feltöltött szöveg
        for (i = 1; getwrd(inbuf, i, wrdbuf) > 0; )
            call putwrd(wrdbuf)
    return
    end
```

A **format** szubrutin hívási fája, a bővítések után a következő:

```
format
    init
    getlin
    comand
        comtyp
        getval
        set
        gettl
        space
        brk
            put
```

```
text
    leadbl
        brk
    underl
    center
        width
    put
        phead (pfoot)
            skip
                putc
            puttl
                putc, putdec
        putc
        putlin
        skip
    getwrd
    putwrd
        width, spread, brk
    space
        brk, phead, pfoot, skip
space
```

Gyakorlatok

7.10 Vizsgáljuk meg a **format** viselkedését, ha valaki visszalépéssel és alsó kötőjelekkel húz alá szavakat. Mi történik a szóközön keresztül való visszalépéssel feltöltött szövegben? Mi a helyzet visszalépés-karaktert tartalmazó szavak aláhúzásával? □

7.11 Az aláhúzott írásjeleket általában nehezebb elolvasni. Módosítsuk az **underl**-t úgy, hogy csak betűket és számjegyeket húzzon alá! □

7.12 Némely terminálon az aláhúzás törli a karaktert, ezért célszerű volna a kiírást

UNDERLINE BACKSPACE *karakter*

sorrendben végezni. Így legalább a karakter megmarad, ha az aláhúzás törlődik is. Módosítsuk az **underl**-t ennek megfelelően! □

7.13 A karakterenként való aláhúzás a lehető legkellemetlenebb anyagot adja a 2. fejezetben szereplő **overstrike** számára. (Ugyanakkor egyes terminálokat is nagyon igénybe vesz.) Az **underl**-t kell-e ügyesebbre alakítani vagy az **overstrike**-ot tökéletesíteni? □

7.14 Az aláhúzás egy másik megoldása az aláhúzandó sorozat két különleges „aláhúzás kezdete", az „aláhúzás vége", zérus szélességű karakter közé való zárása. Elemezzük ezen megoldás előnyeit és hátrányait! □

7.9 MÉRÉSI ADATOK

A **format**-ot több hosszú szöveggel futtattuk és megnéztük, hol tölti az idejét. Az alábbiakban a jelen fejezet formázásával kapcsolatos mérési adatokat közöljük. Akkor a fejezet 2300 sorból, és 9200 szóból állt, melyek közül 640 sor volt formázási parancs; az 1-es sorközzel nyomtatott sornyomtató lista 27 oldalt tett ki.

Mint mindig, a központi egység idejének legnagyobb részét az általunk használt rendszeren (Honeywell 6070) is a B/K műveletek végrehajtása tette ki: az idő 70%-a telt el a **getlin**-ben és alárendeltjeiben (főleg az utóbbiakban), 17%-a a **putlin**-ben és alárendeltjeiben és 2%-a a **putch**-ban. Ezen adatok oly nagy mértékben uralják a futási időt, hogy amíg ezek nem javulnak, a program többi része érdektelen. Tegyük fel azonban, hogy a B/K tevékenység számottevően gyorsítható, ha a szabványos FORTRAN B/K rutinjai helyett karakter bemenetre és kimenetre kihegyezett hatékony rutinokat írunk. A program mely részei válnak ilyen esetben a leginkább időigényessé?

A további jelentősebb rutinok a következők:

getwrd	**3,1%**
putwrd	**2,3**
width	**1,7**
text	**1,6**
spread	**1,0**
put	**0,4**

A B/K idő leszámítása után lényegében ezek a rutinok fogyasztják el a teljes formázási időt. A tanulság most is ugyanaz, de érdemes elismételni. Egy hatékony program létrehozásának legjobb módszere az, hogy keresünk egy jó algoritmust, megírjuk az ezt realizáló, minél áttekinthetőbb programot, majd bemérjük azt. A mérési eredmények közvetlenül elvezetnek ahhoz a néhány rutinhoz, melyeket érdemes úgy módosítani, hogy a lehető leghatékonyabbak legyenek. Ha a rutinokat áttekinthetően írtuk meg, és azok a lokális információt megfelelően elrejtik magukban, módosításuk nem lesz nehéz. Ha az olvashatóságot ennél hamarabb áldozzuk fel a hatékonyság oltárán, akkor, amikor a program nagy részének írása még folyik, nemcsak hiábavaló munkát végzünk, de olyan programot is kapunk, amely nehezen áttekinthető és éppen ezért nehezen is fejleszthető.

7.10 KITERJESZTÉSEK

Mint már említettük, ez meglehetősen alapfokú formázó program, amelyet sok hasznos dologgal lehetne még kiegészíteni anélkül, hogy a kezdő felhasználó számára túlságosan bonyolulttá váljon. Az olyan kiterjesztések, melyek azt teszik lehetővé, hogy a felhasználó több paramétert állíthasson be, annyira egyszerűek, hogy még gyakorlatként sem érdemes közölni őket. Tartalmasabbak, bár kevésbé egyszerűek, az alábbi — a program működését érintő — bővítések.

Egynél több inputfile esete

A jól bevált tervezési elvek szerint a **format** vagy egy file listában szereplő file-okból, vagy ennek hiányában a standard input-ról olvas, akárcsak a 3. fejezetben szereplő **print** program. Célszerű lenne lehetővé tenni egy file tartalmának beillesztését egy formázási parancs segítségével.

Gyakorlatok

7.15 Tegyük lehetővé, hogy a program több paraméterként megadott inputfile-t tudjon kezelni és hogy képes legyen értelmezni egy file befogadási parancsot. Használhatjuk a **.so** *(source)* parancsnevet a „forrás" jelölésére; a

.so *file-név*

parancs hatására a *file név* tartalma bekerül az inputba azon a ponton, ahol a parancs áll. Ha az Olvasó úgy döntene, hogy ez a név jobb, mint a **.include,** hogyan kell módosítani a **command**-ot és a **comtyp**-et? Ha **.so** már működik, a **format**-ot olyan jellegű formázásra is használhatjuk, mint amilyet a 3. fejezetben szereplő **print** végez. Érdemes-e ezután a **print**-et mégis használatban tartani? □

7.16 Vezessük be a $+m$ és $-n$ elhagyható paramétereket úgy, hogy hatásukra a kimenet az m-edik oldalnál kezdődjön és az n-edik oldal után érjen véget! Például a

format +10 −20

a 10.-től a 20. oldalakat (mindkettőt beleértve) nyomtatja ki. Mi az ésszerű működés, ha m vagy n kívül esik az anyag által képviselt lapszám-intervallumon? □

Jobb fej- és lábszövegek

Az eredeti fej- és lábszövegek használata néha nehézkes. A

.he / *bal* / *közép* / *jobb* /

alakú parancs azt fejezi ki, hogy a fejléc három részből áll, amelyeket egy (tetszőleges) jel választ el, például a / . A bal oldali rész balra igazított, a jobb oldali rész jobbra, a középrész pedig középre van állítva. Mint eddig is, a # jel a fejlécben mindenütt az aktuális lapszámmal helyettesítődik.

Gyakorlat

7.17 Valósítsuk meg a bővített **.he** és **.fo** parancsokat! Melyik jobb oldali lapszél határozza meg a részek elhelyezését? Mi történjen, ha a részek átfedik egymást? Hogy lehetne megengedni többsoros fej-, illetve lábszövegeket? Dolgozzuk ki a

.eh .ef .oh .of

parancsokat abból a célból, hogy különböző fej- és lábszöveget írhassunk a páros (**.eh** és **.ef**) és a páratlan (**.oh** és **.of**) sorszámú oldalakra. □

Jelátalakítás

Kivitelkor gyakran szükség van egy-egy karakternek egy másikkal való helyettesítésére. Két kézenfekvő példa: bizonyos karaktereket célszerű lehet szóközre átalakítani, konvertálni, hogy a feltöltött szövegben a szavak egyben maradjanak, vagy egyes karaktereket ponttal helyettesíteni az első pozícióban, hogy formázásról szóló dokumentumokat is leírhassunk.

Gyakorlat

7.18 Valósítsuk meg a **.tr** *(transliterate)* jelátalakító parancsot: a

.tr ab cd

kivitelkor **a**-t **c**-vel, **b**-t pedig **d**-vel helyettesíti;

.tr a—z A—Z

a kisbetűket nagybetűre cseréli ki.

.tr x

az **x**-et szóközzel helyettesíti. Mi történik a következő kombináció esetén?

.tr a b
.tr b c

Átvehető-e a **translit**-ból egy tervezési ötlet vagy egy programrész? □

A „kell még" parancs

Néha célszerű egy programot vagy táblázatot úgy kivinni, hogy az okvetlenül egyetlen oldalra kerüljön. Például, a jelen könyv programjaival is ez történt. A táblázat elején egy „lapkezdet" parancsot kell szimulálni, de csak akkor, ha a táblázat átnyúlna a következő lapra is. A

.ne *n*

(need) parancs jelentése a következő: „még *n* sornyi helyre van szükségem; ha ezen a lapon nincs annyi, kezdjünk új oldalt".

Gyakorlatok

7.19 Valósítsuk meg a **.ne** parancsot. Hogy lehet alkalmazni a parancsot a lap elején vagy végén keletkező elszigetelt sorok kiküszöbölésére? □

7.20 A felhasználók számára roppant kellemetlen ha a **.ne** parancs kiadása előtt sorokat kell számlálnunk. Tervezzünk és dolgozzunk ki módszert arra, hogy egy sorokból álló csoportot együtt lehessen tartani, a megszámlálásuk nélkül. Terjesszük ki a módszert a **.ce** és **.ne** parancsokra is, melyeknél szintén fennáll ugyanez a probléma! □

Több szóköz a mondatok után

Sokan szeretik, ha egy mondatot befejező pontot eggyel több szóköz követ; ez javítja a szöveg külalakját.

Gyakorlat

7.21 Valósítsuk meg ezt a lehetőséget. Győződjünk meg arról, hogy jól működik, ha a pont egy sor végére kerül! Milyen más karakterek és karaktersorozatok fejeznek be mondatokat? Van-e elfogadható algoritmus arra, hogy ne kerüljön többletszóköz monogramokban, rövidítésekben stb. szereplő pontok után? □

Automatikus nagybetűhasználat

Egyes számítóközpontokban vannak kis- és nagybetűs sornyomtatók, de nincs vagy kevés a kétféle betűtípusú terminál vagy lyukasztó eszköz. Ez még nem kell hogy elvegye valakinek a kedvét attól, hogy tetszetős formátumú nyomtatott szöveget hozzon létre.

Gyakorlat

7.22 Módosítsuk a **format**-ot úgy, hogy egyszerű esetekben felfedezze a mondatok végét (ilyen például egy sorvégi vagy egy olyan pont, melyet egy vagy több szóköz követ), és ilyenkor következőként nagybetűt vigyen ki. Egy felhívójelre

is szükségünk lesz, amely felfüggeszti az alapfeltételezést, és a következő karaktert explicit módon alsó vagy felső állásúvá teszi. Ha a mondatok felismerését elég jól végezzük, erre valószínűleg ritkán lesz szükség. Mi lesz ezeknek a karaktereknek a szélessége? □

7.11 LÉNYEGESEBB BŐVÍTÉSEK

Az itt szereplő javaslatok komolyabb munkát igényelnek, ha jól akarjuk őket megvalósítani, de érzékelhető, hogy milyen jelentősen növelhetik a formázó hatékonyságát. Ezen lehetőségek mindegyike — valamilyen formában — rendelkezésre állt a jelen könyv előkészítésekor használt formázóban.

*Elválasztás**

A **format** annyi szót helyez el egy sorban, amennyi csak elfér. Az elválasztás növeli az egy sorba eső szavak átlagos számát, ezért javítja a kimenet külalakját. Feladatunk az, hogy elfogadhatóan jól működő algoritmust találjunk angol szavak programozott elválasztására.

Először is el kell készíteni azon ragok listáját, amelyek jó elválasztási lehetőséget adnak (pl. -tion, -ly stb.). (Gondoljunk arra, hogy angolban szótagokat kell elválasztani, tehát mindkét rész kell hogy magánhangzót tartalmazzon.) Ha csupán ezeket választjuk le a szavakról, már használható lehetőséghez jutottunk. Próbálkozhatunk az igekötők leválasztásával is. Ezzel kapcsolatban azt tapasztaltuk, hogy kevésbé hatásos. Következőként említhető, hogy bizonyos betűpárok soha nem választhatók el (pl. *qu* a legkézenfekvőbb), míg mások, pl. a kettős betűk, majdnem mindig igen. Építsünk fel egy (26×26)-os mátrixot (minden elemre elégséges egy bit), melynek elemei azt mutatják, hogy az egyes betűpárok elválaszthatók vagy nem!

Végezzünk kísérleteket a fenti módszerekkel! Eszébe jut-e az Olvasónak más megközelítés is? Milyen programokat írnánk eszközként ezen feladat megoldásához? Milyen, ide vonatkozó eszközök szerepeltek már a könyvben?

Makrók

Bár a következő fejezetet szenteltük a makrófeldolgozásnak, mégis megemlítjük azt a lehetőséget, hogy a formázót — ha korlátozott módon is — makrókkal bővítsük. Első lépésként bevezetjük a gyakran előforduló parancssorozatok rövidített írásmódját. Például, a

```
.de pp
.sp
.ti +5
.en
```

sorozat egy új parancsot, a **.pp**-t (paragraph) definiálja, egy paragrafus kialakítása céljából. Ezután valahányszor a

```
.pp
```

* Az itt szereplő, az angol nyelvre vonatkozó elválasztási szabályok megfelelői a magyarban is léteznek, de természetesen a két nyelvben nem azonosak *(A fordító)*.

előfordul, az a definiáló szöveg helyettesíti, amely a **.de** és a **.en** között áll. Az eredmény ebben az esetben egy üres sor, és egy ideiglenes bekezdési mélység. Természetesen lehetőség nyílna bármely meglevő parancsnév — mint pl. **.sp** — újradefiniálására is.

Ez idáig egyszerű. A következő lépés a makrók paraméterezhetőségének megvalósítása, hogy különböző paraméterekkel híva, különböző eredményt adhassanak. Például egy címmakrót a következőképpen definiálhatunk:

```
.de tl
.he $1. Fejezet
.en
```

A **$.** szimbólum azt jelenti, hogy a makrót hívásakor az *első* paraméterrel kell helyettesíteni. Ennek értelmében, egy lehetséges hívás:

```
.tl 7
```

amely a „7. Fejezet" címet állítja elő:

Ez a szintaxis a paraméterek számát 9-re korlátozza, ennyi azonban valószínűleg elegendő. A kényelmes használhatóság érdekében a hiányzó paramétereket üres karaktersorozattal kell helyettesíteni, a felesleges paramétereket pedig figyelmen kívül kell hagyni. Hasonló kérdésekre még visszatérünk a 8. fejezetben.

Feltételes formázás

Egy igazán magasszintű formázónak meg kell tudnia változtatni formázási tevékenységét, a futás közben kialakuló feltételeknek megfelelően. Egyik lehetőség erre az

.if *(feltétel) tevékenység*

alakú parancs, mellyel egy feltétel dinamikus vizsgálata és teljesülése esetén a megfelelő tevékenység végezhető el. Ez tetszőlegesen bonyolult lehet, de arra mindenképpen szükség van, hogy a fontosabb paraméterértékek — mint pl. az aktuális oldalszám, a sor- és oldalhosszúság, az aktuális sor szerinti pozíció egy lapon belül, és az, hogy töltési üzemmódban dolgozunk-e — lekérdezhetőek legyenek. Szükségünk lesz aritmetikára, szöveges és numerikus változókra, valamint karaktersorozatokat összehasonlító műveletekre. Például a jelen könyvben szereplő alpontok és gyakorlatok számozását a formázó automatikusan végezte, aritmetikai változók és műveletek segítségével.

Annak, hogy formázónk elegendően hatékony-e, egyik legjobb próbája, hogy lehet-e vele olyan lábjegyzet-készítést végezni, melyben egy oldalon több lábjegyzet is lehet, és ezek szükség szerint további lapokon is folytathatók? Egy másik kritérium, hogy van-e lehetőség többhasábos kimenet készítésére. Ha ezek a tevékenységek viszonylag kényelmesen elvégezhetők, a formázó lényegében minden formázási feladat ellátására alkalmas.

A fenti javaslatok szándékosan elnagyoltak, hogy egyik irányban se befolyásolják túlzottan az Olvasót. Mint mindig, ha meg akarunk valósítani valamit, biztosítsuk annak eszmei integritását — ne csupán egymástól független lehetőségek halmazát hozzuk létre. A megvalósítást pedig lépésenként, és ne egyszerre hajtsuk végre.

BIBLIOGRÁFIAI MEGJEGYZÉSEK

A **format** alapgondolatát J. SALTZER CTSS-beli Runoff programja adta. A Runoff-nak számos változata létezik; a miénk legjobban a Roff-hoz kapcsolódik, amely M. D. MCILROY munkája. Sok formázóprogram kapható kereskedelmi forgalomban, gyakran, mint az ún. „szövegfeldolgozó rendszerek" egy része. Érdekes lehet ezek közül néhánynak a **format**-tal való összevetése.

Ezt a könyvet egy Troff nevű programmal szedték, melyet J. F. OSSANA írt a UNIX-rendszer számára. Egy szedőgép lehetőségei sokkal bővebbek egy sornyomtatóénál: különböző betűtípusok és méretek, jóval bővebb karakterkészlet. Érdekes feladat olyan nyelv tervezése, amely hozzáférést biztosít ezen kibővített lehetőségekhez, de ugyanakkor nem bonyolítja el feleslegesen az alkalmazást a tapasztalatlanabb felhasználók számára. Nagymértékben támaszkodtunk a Troff makró lehetőségeire hogy a formázás részleteit makróutasításokba tömörítsük, melyek könnyen módosíthatók anélkül, hogy a szöveghez hozzányúlnánk.

8. fejezet

Makrófeldolgozás

A makrók valamely alapnyelv bővítésére valók — egyik nyelvről egy másikra való fordítást hajtanak végre. Például, sok programunkban fordul elő olyan sor, mint

while (getlin(buf, infile) ¬= EOF)

ahol **EOF** egy meg nem határozott érték, amely a file végét jelzi. Az olyan „szimbolikus konstansok", mint az **EOF,** így tudják kifejezni egy szám jelentését, ahogyan azt maga a szám sohasem tudná megtenni. Ha egy önkényesen választott számot, pl. (–1)-et használtunk volna, ennek jelentését a környezet ismerete nélkül nem értettük volna meg. Mellesleg az **EOF** értéke gépenként más és más lehet, de sokkal könnyebb és biztonságosabb egy szimbolikus konstans értékét újra definiálni egyetlen helyen, mint egy teljes programon végigmenni és megkeresni az összes olyan (–1)-et, amely ténylegesen a file végét jelzi.

Olyan programra van tehát szükségünk, amely lehetővé teszi az **EOF**-hez hasonló szimbolikus konstansok definiálását oly módon, hogy a név későbbi előfordulásait az őt definiáló karaktersorozat helyettesítse. Az ilyen definíciót hívják *makrónak*, a helyettesítési eljárást *makrókifejtésnek,* a program pedig, amely ezt végrehajtja, a *makrófeldolgozó*. A makrófeldolgozó beolvassa a forrásfile-t és abból egy másik file-t készít, amelyből hiányzanak a makródefiníciók, a makróhivatkozások helyett pedig azok kifejtése áll. Ez lehetővé teszi, hogy olyan helyeken is paramétereket használjunk, ahol egy fordítóprogram konkrét értékeket követelne meg, mint pl. a tömbméret-deklarációban.

Jelen fejezetben első lépésként a **define** nevű programmal foglalkozunk. Ez egy karaktersorozatot egy másikkal helyettesít, ami a makrófeldolgozás legegyszerűbb formája. A programot behívhatjuk például a következőképpen:

define (EOF, –1)

ami után minden **EOF** helyébe (—1) kerül. Bár ez még nem „nyelvi fordítás", de már megkönnyíti a programok olvasását és módosítását. A makrófeldolgozásnak ezt a rutinját használtuk a könyvben előforduló összes programban, ami valamivel hatásosabb, mint a FORTRAN egyes tájszólásaiban előforduló „paraméter" utasítás. Ennél többre nem nagyon volt szükségünk, és a FORTRAN-hoz is olyan közel akartunk maradni, hogy a program érthető legyen.

A következő szint, ami már jóval nagyobb munkát igényel, egy olyan makrófeldolgozó létrehozása, amely paraméteres makrókat is megenged, mint például a

define(putc(c), putch(c, STDOUT))

amely a **putc(x)** összes előfordulását a putch **(x, STDOUT)** utasítással helyettesíti, függetlenül attól, hogy mi az **x** paraméter értéke.

A harmadik szint a makrófeldolgozónak olyan beépített műveletekkel való kibővítése, amelyek kézzelfogható módon megkönnyítik bonyolult makróműveletek megírását. Ezek közül a legfontosabb a feltételvizsgálatok lehetősége. A két utóbbi szint tulajdonképpen már luxus: használatuk kényelmes, megvalósításuk tanulságos, de nélkülük is sok minden megoldható.

Hangsúlyozni szeretnénk, hogy ez nem az egyetlen módja a makrók megadásának. A mi jelölésünk *funkcionális,* vagyis a függvényhivatkozások sok nyelvben használatos írásmódjára emlékeztet, ezért a makróhívások jól beillenek ezekbe a nyelvekbe. Használhattunk volna olyan szintaxist is, mint:

define EOF —1

de ezt nem lehet jól kiterjeszteni a többsoros vagy paraméteres makrók, valamint néhány más beépített művelet irányában, amelyeket pedig szándékunkban áll megvalósítani. A 7. fejezetben a következő formát javasoltuk:

define *név*
törzs
endmarker

amely olyan nyelvek számára megfelelő, ahol a bemenetet soronként kezelik. Egy másik lehetőség a *makrosablonok* feldolgozására, ahol a makrók a műveleteknek felelnek meg (mint a + és a — az aritmetikai kifejezésekben), a paraméterek pedig az operandusok. A makrósablon-feldolgozók használata néha egyszerűbb, megvalósításuk azonban számottevően nehezebb. A fejezet végén levő bibliográfiai megjegyzésekben idevágó irodalmat is javasolunk.

8.1 EGYSZERŰ SZÖVEGHELYETTESÍTÉS

Kezdjük a legegyszerűbb esettel. Másolni akarunk a bemenetről a kimenetre, kivéve bizonyos karaktersorozatokat, amelyeket megjelenésükkor egy előre definiált szöveggel kívánunk helyettesíteni. Az olyan programozási nyelvekben, mint a FORTRAN vagy a PL/1, a helyettesítés természetes egysége valószínűleg az azonosító, vagyis olyan alfanumerikus jelek sorozata, melynek mindkét oldalán nem alfanumerikus karakterek állnak. A

while (getlin(buf, infile) ¬= EOF)

szövegben pl. az **EOF** mindkét oldalán nem alfanumerikus jel áll, ezért elvileg helyettesíthető. Természetesen a **while, getlin, buf** és **infile** esetén ugyanez a helyzet, de mivel ezek feltehetően nincsenek definiálva legalábbis a makrófeldolgozó számára, változtatás nélkül kell őket átmásolni.

A helyettesítés egységét *szimbólumnak (token)* nevezzük. Más esetben egy szimbólum lehet két „nem látható karakter” (szóköz, tabulátor, új sor) közötti karaktersorozat, mint a **format**-ban, vagy lehet adott kezdő és végjel közé zárt szövegrész is. Minden esetben a feldolgozóprogramhoz tartozik egy olyan rutin, amely beolvassa, és valamilyen szabály szerint szimbólumokra bontja a bemenetet.

Hogyan adjuk meg e definíciókat? Az előbb javasolt szintaxis megfelelő: a

define *(név, helyettesítő szöveg)*

a *név* definíciójaként egészen a jobb oldali zárójelig terjedő szöveget határozza meg. Ily módon a helyettesítő szöveg hosszabb lehet egy sornál. Szükségünk lesz olyan modulokra, melyek kigyűjtik a neveket és a helyettesítő szövegeket, valamint olyanokra, amelyek megjelenéskor tárolják az új neveket és azok definícióit.

A megvalósítás egyes részletei igen fontosak, mivel a műveletek végrehajtási sorrendje nagymértékben befolyásolhatja a makrófeldolgozó hatásosságát és használati kényelmét. Az egyik fontos döntés annak meghatározása, hogy mi történjen akkor, amikor egy nevet egy másik név segítségével definiálunk. Például a

define (x, 1)
define (y, x)

definíciók esetén mi történjék *y* beolvasásakor: *x* vagy 1 helyettesítődjék? Vagy

define (y, x)
define (x, 1)

esetén mit kell *y* helyére tenni?

És ezek nem álproblémák! A 3. fejezetben szereplő **archive** programban például van két definíció:

define (DEL, LETD)
define (LETD, 100)

ahol a 100 történetesen a *d* betű belső ábrázolása az ASCII jelkészletben. Jobb a **DEL**-t a **LETD** segítségével definiálni, mint a *d* betűnek egy adott kódrendszer szerinti ábrázolását mereven beépíteni a programba. Mint már említettük, minél kevésbé függ egy program explicit módon a rendszertől, annál könnyebben vihető át egyik rendszerből a másikba, és nagyobb eséllyel éli túl az aktuális rendszer módosításait. Nem akarjuk azt sem, hogy a **define** felhasználóinak túl sokat kelljen törődniük a definíciók sorrendjével. Ennek megfelelően a **define**-t úgy építettük fel, hogy a fentiekhez hasonló példák a lehető legjobban használható eredményt adják — egy makró kiértékelése után a helyettesített szöveg *újra elemzésre kerül.* Ha ez további makrókat tartalmaz, ezek újra a fenti kifejtési eljáráson mennek keresztül. (Ez természetesen végtelen ciklust eredményezhet, amire a későbbiekben fel kell majd készülnünk.)

Több lehetőség van a makróhívások kiértékelésének időbeli meghatározásával kapcsolatban is. Ha egyszer már definiáltuk *x*-et a

define (x, 1)

segítségével, amikor a

define (y, x)

sorra kerül, *x*-et vagy azonnal 1-gyel helyettesíthetjük, vagy figyelmen kívül is hagyhatjuk azt a tényt, hogy az *x* makró, és helyettesítését később is elvégezhetjük, akkor, amikor *y* hívásra kerül. A fenti példában ez a két módszer azonos eredményhez vezet, de különböző lenne az eredmény, ha *x*-et közben újra definiálták volna. A különböző lehetőségek itt némiképpen különböző, de egyformán hasznos makrófeldolgozóhoz vezetnek. A **define** általunk létrehozott megvalósítása a definíciókat nem elemzi makróhívások felderítése céljából, amikor azokat átmásolja a definíciók táblázatába; a makrók értelmezését, amennyire csak lehet, késleltetjük.

A paraméter nélküli makrófeldolgozó vázlata a következő*:

* A *token* és a *szimbólum* szavakat ebben az összefüggésben szinonimáknak kell tekintenünk *(A fordító).*

```
while (gettok(token) ¬= EOF) {
    a szimbólum keresése a táblázatban
        if (token == "define")
            új szimbólum és érték beültetése
        else if (a szimbólumot megtaláltuk)
            az input átkapcsolása a szimbólum definíciójára
        else
            a szimbólum kivitele
            }
```

A többszörös inputforrások miatt ez elvileg rekurzív folyamat is lehet. A **define**-ban a rekurziót nem vermekkel és kapcsolt listákkal kezeljük, mint más programokban. Erre rövidesen visszatérünk.

A **gettok** hasonló a 3. és 7. fejezetekből már ismert **getwrd** rutinhoz, de annál annyival bonyolultabb, hogy nemalfabetikus jelek megfelelő kezelésére is képes. Például, a szóközök is értékes karakterek, nem szabad őket kihagyni. A

```
t = gettok(token, maxtok)
```

hívás a következő szimbólumot a standard inputról **token**-be másolja. Egy szimbólum vagy egy betűkből és számokból álló jelsorozat vagy egyetlen nem alfanumerikus jel. A **gettok** által visszaadott érték egy **ALPHA** (az alfanumerikus szimbólumok esetén), vagy maga a karakter (a nem alfanumerikus esetben). (Ebből következik, hogy **ALPHA** szükségképpen különbözik minden nem alfanumerikus jeltől.) A **gettok** a **type**-ot használja, melyet a 4. fejezetben írtunk meg, annak megállapítására, hogy egy karakter betű-e, számjegy-e vagy valami más. Emlékezzünk vissza, hogy nem alfanumerikus típus esetén a **type** magát a karaktert adja vissza.

```
# gettok — alfanum. sorozat vagy egyetlen nem alfajel olvasása a define
    # számára
    character function gettok(token, toksiz)
    character ngetc, type
    integer i, toksiz
    character token(toksiz)

    for (i = 1; i < toksiz; i = i + 1) {
        gettok = type(ngetc(token(i)))
        if (gettok ¬= LETTER & gettok ¬= DIGIT)
            break
        }
    if (i >= toksiz)
        call error("túl hosszú szimbólum")
    if (i > 1) {        # alfanumerikus jelet találtunk
        call putbak(token(i))
        i = i - 1
        gettok = ALPHA
        }
    # a szimbólum különben egykarakteres
    token(i+1) = EOS
    return
    end
```

Amikor a szimbólumokat karakterenként szedjük össze, egy szimbólum végét csak akkor ismerjük fel, ha egy karakterrel már túlmentünk rajta. Ez a nemkívánatos mellékhatás klasszikus példája, amely, ha engedjük elszabadulni, csúnyán szétzilálhatja a programot. Ugyanis valahányszor a következő karakterre lenne szükségünk, ellenőriznünk kell, hogy szükséges-e beolvasnunk egy új karaktert vagy használhatjuk a már bent levőt. Ha ez összekeveredik az egyes karakterek esetén elvégzendő tevékenységekkel, az eredmény olvashatatlan zagyvalék lehet.

A bonyodalmakat úgy kerüljük meg, hogy bevezetünk egy együttműködő rutinpárt. Az **ngetc** a következőként megvizsgálandó bemenő jelet adja eredményül mind paraméterének, mind pedig saját függvényértékének. A **putbak** rutin pedig egy karaktert visszaküld inputnak oly módon, hogy a következő **ngetc** hívás újra ezt a karaktert hozza be. Ezután valahányszor a **gettok** eggyel több karaktert olvas mint kellene, ezt azonnal visszaküldi, és így a program többi részének nem is kell tudnia a problémáról.

Elvileg a **putbak** elemi művelet kellene, hogy legyen, az **ngetc** pedig egyszerűen a **getc** szerepét játszaná. Azért választottuk szét őket, hogy bemutassuk, miként lehet az ilyen visszaküldést elvégezni, mivel általában az Olvasónak majd saját változatát kell előállítania. A **putbak** a visszaküldött karaktereket egy pufferbe teszi. Az **ngetc** a pufferból olvas, ha az nem üres, és a **getc**-et hívja, ha a puffer üres.

```
# putbak — egy karakter visszaküldése az input-ra
   subroutine putbak(c)
   character c
   include cdefio

   bp = bp + 1
   if (bp > BUFSIZE)
         call error("a visszaküldött jelek száma túl nagy")
   buf(bp) = c
   return
   end

# ngetc — egy (esetleg már visszaküldött) karakter olvasása
   character function ngetc(c)
   character getc
   character c
   include cdefio

   if (bp > 0)
         c = buf(bp)
   else {
         bp = 1
         buf(bp) = getc(c)
         }
   if (c ¬= EOF)
         bp = bp - 1
   ngetc = c
   return
   end
```

A **bp** a pufferből kiveendő következő karakter indexe. Ha a **bp**-ben nulla van, a **getc** hívásával egy új karaktert hozunk be. (**bp**-t kezdetben nullára kell állítani.) Az **ngetc** és **putbak** által használt puffer és mutató a **cdefio** nevű közös adatmezőben helyezkedik el:

```
common /cdefio/ bp, buf(BUFSIZE)
    integer bp          # soronkövetkező karakter; kezdőért.=0
    character buf       # a visszaküldött karakterek
```

A **gettok** sohasem küld vissza egynél több jelet az **ngetc** két hívása között, ezért a **buf** tömb helyett közönséges skalár változó is lehetne. Azonban a visszaküldés hasznos művelet, amely megéri az általánosítást. Írhatnánk egy **pbstr** (push back string) nevű rutint is, amely a **putbak** ismételt hívásaival egy hosszabb jelsorozatot küld vissza.

```
# pbstr — egy jelsorozat visszaküldése az inputra
    subroutine pbstr(in)
    character in(MAXLINE)
    integer length
    integer i

    for (i = length(in); i > 0; i = i - 1)
        call putbak(in(i))
    return
    end
```

A jelsorozatot természetesen fordított sorrendben kell visszaküldeni.

A **cdefio** közös adatmezőt csak az **ngetc** és a **putbak** használja. A **pbstr** gyorsabb lehetne, ha szintén használná a **cdefio**-t, és nem lenne kénytelen minden egyes jel visszaküldésekor a **putbak**-ot behívni, de amennyire csak lehetséges, igyekeztünk a rutinok közötti adatkapcsolatokat a minimálisra csökkenteni. Ez az általunk ismert egyik leghatékonyabb módja a könnyen módosítható programok írásának. Természetesen, ha később bebizonyosodik, hogy a **pbstr** lassú működése szűk keresztmetszetet hoz létre, tudnánk rajta gyorsítani. A fontos az, hogy jó terv alapján induljunk, mert sokkal könnyebb egy jól megírt programban alkalmazott elvekből engedni, mint egy rossz programra az elveket később ráerőszakolni.

Mivel mást is visszaküldhetünk, mint amit olvastunk, felmerülhetett már az Olvasóban az ötlet, hogy egy kifejtett makrószöveg újraelemzése a **putbak** segítségével elegánsan megoldható. Tegyük fel, hogy ha egy definiált nevet találunk, a neki megfelelő *helyettesítő szöveget* küldjük vissza a bemenetre. Amikor ezt beolvassuk és a visszaolvasott szöveg szintén tartalmaz egy definiált nevet, ezt megkeressük és éppúgy helyettesítjük, mintha az eredeti bemenetben fordult volna elő. Ezen visszaküldés segítségével kezeljük a többszörös forrásokból való bevitelhez tartozó implicit rekurziót.

Most már megírhatjuk a **define** főprogramját:

```
# define — egyszerű jelsorozat-helyettesítéssel működő makrófeldolgozó
    character gettok
    character defn(MAXDEF), t, token(MAXTOK)
    integer lookup
```

```
        string defnam "define"
        integer deftyp(2)
        data deftyp(1) /DEFTYPE/, deftyp(2) /EOS/

        call instal(defnam, deftyp)
        for (t = gettok(token, MAXTOK); t ¬= EOF; t = gettok
                (token, MAXTOK))
            if (t ¬= ALPHA)          # nem alfa szimbólumok kivitele
                call putlin(token, STDOUT)
            else if (lookup(token, defn) == NO)     # valamint a
                                                    # nem definiáltaké
                call putlin(token, STDOUT)
            else if (defn(1) == DEFTYPE) {          # a definíció elővétele
                call getdef(token, MAXTOK, defn, MAXDEF)
                call instal(token, defn)
                }
            else
                call pbstr(defn)   # a helyettesítő szöveg visszaküldése
                                   #   az inputra
    stop
    end
```

Ha a **gettok** nem **ALPHA** típusú szimbólumot ad vissza, akkor az nem lehet definiált szimbólum. Ezt azonnal megvizsgáljuk, hogy ezáltal elkerüljük az összes nem alfanumerikus jel ellenőrzését.

A szimbólumot, ami ez esetben egy név, a **lookup** rutin keresi a definíciók között. Ha megtalálta, a definíció szövegét adja vissza. Ha a **lookup** a szimbólumot nem találta meg, az azonnal kivihető, speciális jelentése nincs. Ha a szimbólum a **define,** a nevet és a helyettesítő szöveget a **getdef** rutin szétválasztja, az **instal** rutin pedig elhelyezi a táblázatban. Ha a név szerepelt a táblázatban, de nem **define** volt, akkor a megfelelő helyettesítő szöveget visszaküldjük az inputra.

Az **instal** szerepe az, hogy a táblázatban elhelyezze a **define** kulcsszót és ennek fordításaként a **DEFTYPE** szimbolikus konstansot. Ez jobb megoldás, mintha ugyanezt egy **data** utasítással végeznénk el, mivel így a programnak nem kell ismernie a táblázatbeli bejegyzések formáját, a táblázat kezdeti tartalmának beállítása pedig *sokkal* egyszerűbb, mivel azt csupán üres állapotba kell hoznunk. A **lookup** a fordítást adja vissza, ha **define**-t talál, így gyorsan ellenőrizhető, hogy egy név a **define**-e. A **DEFTYPE** természetesen minden más karaktertől különbözik. A táblázatkezelő algoritmus látható része csupán a **lookup**-ból és az **instal**-ból áll, ezekkel a következő pontban foglalkozunk.

A **getdef** programja a következő:

```
# getdef (paraméter nélküli eset) — név és definíció feldolgozása
    subroutine getdef(token, toksiz, defn, defsiz)
    character gettok, ngetc
    integer defsiz, i, nlpar, toksiz
    character c, defn(defsiz), token(toksiz)
```

```
if (ngetc(c) ¬= LPAREN)
    call error("bal zárójel hiányzik")
else if (gettok(token, toksiz) ¬= ALPHA)
    call error("nem alfanumerikus név.")
else if (ngetc(c) ¬= COMMA)
    call error("define-ban egy vessző hiányzik.")
# különben "(név," leválasztása
nlpar = 0
for (i = 1; nlpar >= 0; i = i + 1)
    if (i > defsiz)
        call error("túl hosszú definíció.")
    else if (ngetc(defn(i)) == EOF)
        call error("hiányzó jobb zárójel.")
    else if (defn(i) == LPAREN)
        nlpar = nlpar + 1
    else if (defn(i) == RPAREN)
        nlpar = nlpar - 1
    # különben defn(i) normál karaktert tartalmaz
defn(i-1) = EOS
return
end
```

A munka legnagyobb részét itt a zárójelek egyensúlyának figyelése és az input hibáinak kezelése teszi ki.

Figyeljük meg, hogy az **else**-ek legnagyobb részét nem kellene kiírnunk, mivel az **error** nem adja vissza a vezérlést a hívónak. Ennek ellenére kiírtuk őket annak hangsúlyozására, hogy a tevékenységsorozatokból legfeljebb egy kerül végrehajtásra. A programot az Olvasó gyorsabban megérti, ha szokványos szerkezeteket lát, ezek közül pedig az **else if** sorozat az egyik legfontosabb.

Az **else if** ilyen alkalmazása analóg az **andif** 6. fejezetbeli használatával. Ott egy lépéssorozatot akartunk végrehajtani, minden lépés után állapotvizsgálatot végeztünk és le kellett állni, amikor az első vizsgálat hamis eredményt adott. Itt ugyancsak egy lépéssorozatot hajtunk végre, de akkor akarunk leállni, amikor az első vizsgálat *sikeres*. Ennek értelmében ezt az alkalmazást **orif**-nek is hívhatnánk.

A lényeg az, hogy észrevegyük:
— minden sorrakerülő vizsgálat mellékhatással is járhat;
— a lépések végrehajtásának sorrendje nem közömbös;
— pontosan egy befejező tevékenységsorozatot akarunk végrehajtani attól függően, hogy a vizsgálatok sorozatában milyen messzire jutottunk.

Programjainkban gyakran szerepelnek ilyen szerkezetek.

Gyakorlatok

8.1 Mi történik a

```
define (d, define)
d(a, b)
a
```

bemenő szöveg esetén?
Mi történik a

define (define, x)
define (a, b)

bemenő szöveg esetén? □
8.2 Mi a hatása a

define (x, x)

vagy a

define (x, y)
define (y, x)

definícióknak, *x* szempontjából? Milyen hatást szeretne az Olvasó elérni? □
8.3 Ha egy sorban csupán egyetlen definíció áll, az ezt követő szóközök és az újsor-karakter átmásolódik a kimenetre, bár természetesnek tűnne ezek teljes elhagyása. Módosítsuk **getdef**-et úgy, hogy a pusztán definíciót tartalmazó sorokból ne jöjjön létre kimenő szöveg! Elfogadható ez a megoldás akkor, ha a kimenet egy fordítóprogram bemenete, amely hibajelzéseihez felhasználja a sorszámokat? □
8.4 Az előző probléma egy másik, általánosabb megoldásaként dolgozzuk ki a **dnl** *(delete new line)* beépített műveletet, amely előfordulásától kezdve a következő újsor-karakterig minden karaktert töröl. Ennek értelmében a

define (x, 1) dnl

alapján **dnl** törli a definíció utáni szöveget, a sorból tehát nem keletkezik output. □

8.2 KERESÉS A TÁBLÁZATBAN

Térjünk most rá a **lookup** és az **instal** tervezésére. Ezek a rutinok kezelik a neveket és definíciókat tartalmazó táblázatokat. Egy fontos tervezési döntést már hoztunk — a táblázatok formája, a keresés módszere stb. magánügy, csupán a **lookup** és az **instal** ismeri ezeket. Minden más rutin rajtuk keresztül érheti el a táblázatot. Az információ elrejtése a jó programtervezés szempontjából igen fontos: azon rutinok, melyeknek nem kell ismerniük egy adatszerkezet belső ábrázolását, ne is tudjanak róla. Ez nemcsak azt biztosítja, hogy az adatokat szándékolatlanul ne írjuk felül, hanem — ami ennél fontosabb — a programot egymástól független részekre bontja, melyek egymás zavarása nélkül módosíthatók. Minden rész „fekete doboz", amely kifelé csak egy jól definiált csatlakozási felületet mutat. A mi esetünkben, ha a táblázattal kapcsolatos megoldások közül egyeseket módosítunk — pl. rendezzük a neveket, vagy „hash" módszerrel* tároljuk őket — azt büntetlenül tehetjük, mivel más rutinok nem tudják, hogyan szerveztük meg a táblázatot. Természetesen jól kell felmérnünk azt, hogy egy rutinnak mit kell ismernie. Sokkal könnyebb olyan rutinokat tervezni, amelyek adatszerkezetét az őket hívó programoknak ismernie „kell", bár az gondosabb munkával elrejthető lett volna.

A táblázatban való keresés programján belül, a keresési módszer határozza meg a táblázat szerkezetét. Pillanatnyilag a legfontosabb szempont az egyszerűség. Feltesszük, hogy a definíciók általában szabálytalan időközönként, és nem egyszerre érkeznek. Ez a táblázat elemeinek rendezése és a bináris keresés ellen szól. A „hash"

* A „hash" módszer az, amikor a név táblázatbeli indexét egy rögzített algoritmus alapján határozzák meg, melynek eredménye csak a névtől függ *(A fordító)*.

módszer és a faszerkezetű tárolás kezdetnek mindenképpen túl bonyolult. Ezért az új bejegyzéseket egyszerűen érkezésük sorrendjében a lineáris táblázat végéhez csatlakoztatjuk, és a szimbólumokat a táblázatban lineárisan keressük. Egy program létrehozásának kezdeti szakaszában a magas szintű keresési módszerek alkalmazása nem éri meg a program bonyolultságának növekedését. A lineáris keresés nem mindig a legjobb, de első megoldásként mindenképpen kiváló. Megvalósítása egyszerű, és nagy valószínűséggel első kísérletre működni fog. Ha később szűk keresztmetszetnek bizonyul, egy gyorsabb algoritmussal helyettesíthető, a program más részeinek érintése nélkül.

A lineáris táblázat szerkezete a következő: Egy nagy táblázat egymás után tartalmazza a neveket és a helyettesítő szövegeket:

név EOS definíció EOS név EOS definíció EOS...

Egy másik tömb elsősorban a név bejegyzésekre mutató pointereket tartalmaz.

```
common /clook/ lastp, lastt, namptr(MAXPTR), table(MAXTBL)
    integer lastp        # namptr utolsó eleme; kezdőért.=0
    integer lastt        # table utolsó eleme; kezdőért.=0
    integer namptr       # név-mutatók
    character table      # a nevek és definíciók tényleges szövege
```

A **lastp** és **lastt** rendre **namptr**-ben és a **table**-ben utoljára használt pozíciókra mutatnak, ezeket kezdetben nullára kell állítani. Mindezen változókat a **clook** nevű közös adatmezőben helyeztük el, mivel mind a **lookup**, mind az **instal** használja őket.

A **lookup YES**-t ad vissza és kiemeli a definíciót, ha a szimbólumot megtalálta, különben **NO** a válasz.

```
# lookup — a név megkeresése és a definíció kiemelése a táblázatból
    integer function lookup(name, defn)
    character defn(MAXDEF), name(MAXTOK)
    integer i, j, k
    include clook

    for (i = lastp; i > 0; i = i - 1) {
        j = namptr(i)
        for (k = 1; name(k) == table(j) & name(k) ¬= EOS; k=k+1)
            j = j + 1
        if (name(k) == table(j)) {       # egy megvan
            call scopy(table, j+1, defn, 1)
            lookup = YES
            return
            }
        }
    lookup = NO
    return
    end
```

Az **scopy** a 3. fejezetben leírt, jelsorozatot másoló szubrutin.

Az **instal** egy új nevet és definíciót illeszt a táblázat végéhez; akkor hívjuk, ha **define** fordul elő. Az **instal** nem ellenőrzi, hogy a név benne van-e már a táblázatban. A neveket egyszerűen új definícióval lehet újradefiniálni; mivel a **lookup** a táblázatot visszafelé haladva vizsgálja, az új definíció hatálytalanítja a régit.

```
# instal — név és definíció elhelyezése a táblázatban
    subroutine instal(name, defn)
    character defn(MAXTOK), name(MAXDEF)
    integer length
    integer dlen, nlen
    include clook

    nlen = length(name) + 1
    dlen = length(defn) + 1
    if (lastt + nlen + dlen > MAXTBL | lastp >= MAXPTR) {
        call putlin(name, ERROUT)
        call remark(": túl sok definíció.")
        }
    lastp = lastp + 1
    namptr(lastp) = lastt + 1
    call scopy(name, 1, table, lastt + 1)
    call scopy(defn, 1, table, lastt + nlen + 1)
    lastt = lastt + nlen + dlen
    return
    end
```

Gyakorlatok

8.5 Igazoljuk, hogy a **getdef** és az **instal** jól működik, ha a definíció üres: a

define (semmi,)

egy helyettesítő szöveg nélküli jelsorozatot definiál. Miért lehet ilyesmire szükség? Mi a hatása a

semmi (a szöveg egy sora)

makróhívásnak? □

8.6 A nevek újradefiniálása az előző tárolóhely felszabadítása nélkül kétségkívül pazarlás, ha gyakran csinálják. Hozzuk létre az **undefine** parancsot, amely

undefine (név)

formában megszünteti a név legutóbbi definícióját. Mi történjék, ha az **undefine** paramétere egy nem definiált név? □

8.7 Végezzünk méréseket a **define**-val kapcsolatban, és határozzuk meg, hol tölti az időt. Megéri-e a táblázatban való keresés gyorsítása? Az egyik lehetőség az, hogy a **table**-ban levő neveket minden új definíció érkezésekor rendezzük. Valójában a **namptr**-beli mutatókat kellene rendeznünk, amint ezt a 4. fejezetbeli **sort** esetén tettük. Ezután a **lookup**-ot kellene módosítani, hogy bináris keresést végezzen. Mennyivel lett gyorsabb a program? Mennyire egyszerű ezt és az előző gyakorlatot párhuzamosan megoldani? □

8.8 Valósítsunk meg egy olyan változatot, amely nem használ visszaküldést abban az értelemben, ahogy ezt mi tettük, hanem az aktuális bemeneteket egy ve-

remben tárolja és a megfelelő módon vált át egyikről a másikra. Melyik változat egyszerűbb és melyik a gyorsabb? □

8.9 Hogy birkózik meg a **define** a közönséges programozási nyelvekben használatos megjegyzésekkel? Fel kell-e ismernie a **define**-nak az idézőjelek között álló, közvetlenül felhasználásra kerülő jelsorozatokat? Pontosabban, kell-e az ilyenekben szereplő neveket helyettesítenie? □

8.10 Gyakran hasznos, ha van legalább egy egyszerű feltételvizsgálati lehetőségünk. Tegyük fel, hogy az

ifdef (név, szöveg)

parancs azt jelenti: „ha a név definiált, a szöveg legyen része a bemenetnek, különben hagyjuk ki". Ekkor egy programot különböző gépekre paraméterezhetünk, az alábbiakhoz hasonló definíciók segítségével:

ifdef (gép1, define (szóhossz,32) define (karakterhossz, 8))
ifdef (gép2, define (szóhossz,36) define (karakterhossz, 6))

stb. Ezután **gép1**-et az üres definícióval adjuk meg:

define (gép1,)

A szóhossz, valamint a többi paraméter értéke pedig helyesen állítódik be az **ifdef** sorok előfordulásaikor. Ha ezt az egyetlen sort kicseréljük és a szöveget újra feldolgozzuk, a program átállítható **gép2**-re. Valósítsuk meg ezt a feltételkezelési lehetőséget! □

8.3 NÉHÁNY MÉRÉSI EREDMÉNY

Méréseket végeztünk a **define**-nal néhány „tipikus" RATFOR programon (a fejezetben szereplő programokon), hogy megnézzük, hogyan oszlik meg a gépidő felhasználása. A mérési eredmények a következők:

	# hívások száma	a központi egység ideje %-ban
(read)	472	40,3
(write)	9199	14,4
lookup	1636	13,3
gettok	5263	9,0
getc	9902	6,3
ngetc	12149	4,7
putlin	5092	4,5
define	1	3,6
type	11868	2,7
putbak	2247	0,4
getdef	42	0,2
pbstr	86	0,2
scopy	214	0,2
instal	43	0,1
length	172	0,1

A **(read)** rutin a FORTRAN formázott B/K rendszere segítségével sorokat olvas be. A **(write)** lényegében az 1. fejezetben szereplő **putc**.

Mielőtt a futási idő százalékos megoszlását vizsgálnánk, érdekes megjegyezni, hogy a mérés sok olyan egyszerű információt is nyújt, mint pl. az egyes szubrutinok hívásainak száma, különösen a jól modularizált programok esetén, amikor egy modul csak egyetlen feladatot végez el. Például a fenti adatokból kitűnik, hogy a bemenet 472 sorból és 9902 jelből állt (ezek egyike egy **EOF**), amelyek közül 9199 kimenő jel volt, az 1636 szimbólumból 42 volt definiálva (ami arra mutat, hogy a **lookup** a legtöbbször hiába vizsgálta végig a teljes táblázatot) stb. Ezen adatok némelyike lehetőséget ad a program működési összhangjának ellenőrzésére. Eggyel több **instal** hívásra számítunk, mint a **getdef** hívások száma (mivel első helyen a **define** kulcsszó áll); ha ez nem így történt, valami komoly hiba van. Azt is megfigyelhetjük, hogy nem alfanumerikus szimbólumok keresése majdnem megnégyszerezi a **lookup** hívások számát és jelentősen befolyásolja a központi egység idejét, tehát az a döntésünk, hogy ezt az esetet külön vizsgáljuk, igazolást nyert.

Elvileg az ilyen információt — ha szükségünk van rá — automatikusan kellene kapnunk, de ez gyakran külön ráfordítással sikerül. A 9. fejezetben foglalkozunk az ezzel kapcsolatos tennivalókkal.

A központi egység időadatai — mint ez a FORTRAN-ban szokásos — tanúsítják, hogy a B/K idő vezet, mintegy 65%-kal. A maradék időn belül a táblázatban való keresés a legjelentősebb, de nem kiugró. Természetesen a keresési idő nő a táblázat bejegyzéseinek számával, ezért ha ez lényegesen nagyobb lenne, a keresés végül meghaladná a B/K idejét. A visszaküldési algoritmus karakterenként szerény időtöbbletet jelent, de ennél jóval többet ér a programnak az algoritmus által biztosított áttekinthetősége. Az a döntés, hogy **pbstr**-t a **putbak**-ra építve írjuk, szintén igazolást nyert.

A hatékonyság érdekében hozott döntéseink gyakorlati alátámasztása lényeges. Ennek oka az, hogy a legtöbb programozó nem tudja, hol tölti a program az idejét. Ebből következőleg a „hatékonyság" javítására az (áttekinthetőség feláldozásával) fordított erőfeszítés nagy része hiábavaló. Tapasztalatunk szerint a túl korai optimalizálást a programon végzett rendszeres mérésekkel lehet elkerülni. Csakis ilyen, első kézből származó tapasztalatok adják meg a prioritások helyes felméréséhez szükséges értéket.

8.4 PARAMÉTERES MAKRÓK

A paraméteres makrók nagymértékben növelik a makrófeldolgozó erejét. Például azt mondtuk, hogy a **getc** és **putc** rendre azonosak a **getch(c, STDIN)** és **putch (c, STDOUT)** rutinhívásokkal. Ha a **getc**-t és **putc**-t makróként definiáljuk, melyek kifejtésében **getch**-ra és **putch**-ra való hivatkozások szerepelnek, ezzel biztosíthatjuk az azonosságot és kiküszöbölünk egy szubrutinhívási szintet, ami javíthatja a program hatékonyságát. Ezt a kifejtést paraméterezhetőség nélkül nem lehet elvégezni.

Összetettebb példaként tekintsük a 6. fejezetben írott **skipbl(s, i)** szubrutint, melynek teljes törzse a következő:

```
while (s(i) == BLANK | s(i) == TAB)
    i = i + 1
```

Ez a rutin olyan rövid, hogy makróként is meg lehetne valósítani úgy, hogy kifejtése hívás helyett nyílt formában kerüljön a programba. (Ez akkor lenne célszerű, ha a mérések azt mutatnák, hogy szubrutinként többlettárolót vagy futási időt igényel.) Definiálhatnánk a

```
skipbl (s, i)
```

makrót, amelynek kifejtése a fenti sorokat adná. Az **s** és **i** formális paraméterek helyére minden egyes makróhíváskor azok aktuális értéke helyettesítődne. A felhasználó szempontjából az egyetlen különbség az lenne, hogy a **call** utasítást nem kellene leírni.

A paraméteres makrók definiálásának szintaxisa az eddig használt

define *(név, helyettesítő szöveg)*

név-et definiáló alak kiterjesztése: A $*n* alakú kifejezésnek (ahol *n*=1,2...9) a helyettesítő szövegben való előfordulása a makró hívásakor az *n*-edik paraméterrel helyettesítődik. Ennek értelmében a

```
define(skipbl,while($1($2)==BLANK|$1($2)==TAB)$2=$2+1)
```

definiálja a **skipbl** makrót.

A paraméterek $*n*-nel való jelölése egyszerűbb, bár kevésbé kényelmes, mint a formális paramétereknek nevekkel való megadása az alábbi módon:

```
define(skipbl(s, i), while (s(i) == BLANK | s((i) == TAB) i = i + 1)
```

Állandó alapelvünk, hogy először készítsünk egy áttekinthető, elfogadható és működő programot, majd ha szükséges, ezt később tökéletesítsük.

Ugyanezen elmélet másik megnyilvánulása a paraméterek számának 9-re való korlátozása. Nincs értelme addig sokparaméteres makrókkal bajlódni, ameddig a feldolgozóprogram többi része nem működik. Különben a gyakorlatban majd az is kiderül, hogy ritkán van több paraméterre szükség. A bonyolultabb problémákat félre kell tenni, ameddig az egyszerűbbeken teljes biztonsággal úrrá nem lettünk.

További példaként említhető, hogy sok file kezeléssel foglalkozó programunkban megtalálhatók a következő sorok:

```
fd = open(name, READ)
if (fd == ERR)
    call cant(name)
```

Ha definiáljuk a **copen** makrót, a fenti műveletek egyetlen sorrá zsugorodnak:

```
copen (név, READ, fd)
```

a közös szerkezet pedig tisztábbá válik. A megfelelő makródefiníció a következő:

```
define(copen, {$3 = open($1, $2)
          if ($3 == ERR)
               call cant($1) } )
```

Amikor a

```
copen (név, READ, fd)
```

hívás sorrakerül, **$1** helyére **név**, **$2** helyére **READ** és **$3** helyére **fd** helyettesítődik. A definícióban azért szerepelnek a kapcsos zárójelek, hogy az

```
if (getarg(1, name, size) ¬= EOF)
    copen(name, READ, fd)
```

hívás kifejtése után **copen** törzse egy összetett utasítást alkosson.

A paramétereket elfogadó makrófeldolgozót nehezebb megvalósítani, mint a paraméter nélkülit. Továbbá, a **define**-on kívül be akarunk vezetni néhány további „beépített" műveletet is, a feltételes utasítást, a korlátozott aritmetikát és egy rész-

sorozat funkciót; mindezt pedig húsbavágó változtatások nélkül szeretnénk elérni. Fő célunk annak biztosítása, hogy bármely művelet — makróhívás, definíció vagy egyéb beépített tevékenység — előfordulhasson bármely másik belsejében. Ha ez lehetséges, akkor elvben a makrófeldolgozó bármilyen gépi számítás elvégzésére képes, bár ennek kifejezése esetleg nem könnyű.

Mindaddig, amíg makróhívások (illetve a velük azonosan kezelt beépített műveletek) nem fordulnak elő, a bemenetet közvetlenül átmásoljuk a kimenetre. Makróhívás esetén azonban össze kell gyűjteni a nevét, definícióját és esetleges paramétereit. Ha a paraméterek összegyűjtése befejeződött, a makró kiértékelése a következő: Ha beépített műveletről van szó, mint amilyen pl. a **define**, behívjuk a megfelelő rutint, amely minden, a paraméterekkel kapcsolatos műveletet elvégez. Ha a makró nem beépített művelet, a definíció szövegét visszaküldjük a bemenetre. Visszaküldés közben minden $\$n$ a megfelelő — már összegyűjtött — paraméterrel helyettesítődik.

A folyamat akkor válik érdekessé, amikor a paraméterek egyike egy másik makróhívást vagy beépített műveletet tartalmaz. Többféle megoldás ismeretes, melyek közül a nem rekurzív nyelvek számára az egyik legegyszerűbb a paraméterek összegyűjtéskor való értelmezése és a bemenetre való visszaküldése.

Amikor makróhívást találunk, a név és a definíció egy veremként szervezett kiértékelő területre kerül. A hívásban szereplő paraméterek szintén ide másolódnak, kivéve azokat, melyek egy másik skatulyázott makróhívást tartalmaznak. Ilyenkor a veremben újabb kiértékelő terület keletkezik, ami csak a belső makró teljes kiértékelése és a bemenetre való visszaküldése után szabadul fel, amikor is a külső makró feldolgozása folytatódik. A külső makró soha nem látja a belsőt, csak annak kifejtését. (Természetesen a belső makró még további makrókat hívhat; a folyamat rekurzív.)

Az állandóan szem előtt tartandó elv az, hogy a paraméterek kiértékelése összegyűjtésük közben teljes mértékben megtörténik. Ez más módszer, mint a fejezetben korábban bemutatott jelsorozat-helyettesítési eljárás, de szokványos alkalmazások esetén, mint pl. programok szimbolikus paramétereinek helyettesítése, a két módszer azonos eredményre vezet. (Mivel késleltetett kiértékelésről is gondoskodni fogunk, szükség esetén kihasználhatjuk az előző módszer előnyeit is.) A programok elkészítése előtt lássunk néhány példát.

Tegyük fel, hogy a

```
define (EOS, 0)
```

definíció feldolgozása folyik. Amikor a **define**-t látjuk, **DEFTYPE** és **define** a kiértékelő verem 1. és 2. pozíciójába kerül. (A sorrendet később indokoljuk.) Ezután összegyűjtjük a paramétereket. Pillanatnyilag sem az **EOS**, sem a **0** nem okoz problémát, ezek rendre a verem 3. és 4. pozíciójába kerülnek. Végül elvégezhetjük a kiértékelést, ami ebben az esetben egy rutin behívását jelenti a név és a definíció — azaz a 3. és 4. pozícióban levő paraméterek — táblázatbeli elhelyezése céljából. Ezután a verem négy felső eleme törlődik.

Ha ezután egy **EOS** van a bemenetben, definíciójával és paraméterek nélkül a verembe kerül. A **0** definíciót visszaküldjük az inputra, a verem felső szintje pedig törlődik.

Bonyolultabb példaként tekintsünk egy paraméteres definíciót:

```
define (bump, $1 = $1 + 1)
```

A **bump** tehát a paraméterét eggyel növelő makró. A

```
bump (x)
```

hatására **bump** és **$1=$1+1** átmásolódik a kiértékelő verembe. Az **x**-et mint paramétert leválasztjuk. Ha **x** nem definiált név, akkor **bump** feldolgozása véget ért. A definíciót visszaküldjük a bemenetre, miközben **$1** minden előfordulását **x**-szel helyettesítjük, aminek eredménye **x=x+1**.

Most képzeljük el egy pillanatra, hogy **x**-et korábban már definiálták, mondjuk úgy, hogy legyen **y**. Ekkor **x** egy makróhívás, ezért amikor előkerül, új veremszintet kell nyitni, amelybe beírjuk **x**-et és definícióját. Ezután az új definíció megy vissza az inputra és az új veremszint törlődik. Amikor a paraméterleválasztás folytatódik az előző (**bump**) szinten, a régebbi **x** input már **y**-ná vált, és ez lesz a **bump** tényleges paramétere. Ebből a **bump**-ra annyi tartozik, hogy az **y** paraméterrel hívták, ezért az eredmény **y=y+1**.

Gyakorlat

8.11 Tegyük fel, hogy **getc**-t és **putc**-t a **getch** és **putch** segítségével definiálták. Menjünk végig gép nélkül a

putc (getc(c))

híváshoz tartozó kifejtési eljáráson, beleértve az **STDIN**-hez és **STDOUT**-hoz tartozó értékek feldolgozását is! □

8.5 A MEGVALÓSÍTÁS

A feldolgozást most már részletesebben leírhatjuk:

```
while (gettok token, maxtok) ¬= EOF
  if (type==ALPHA)
        if (lookup (token)==NO)
        a szimbólum közvetlen kivitele vagy átvitele a kiértékelő verembe
        else
        új terület kijelölése a veremben
        a név és a definíció átmásolása az aktuális területre
  else if (a verem üres) # nincs paramétermegőrzés
        közvetlen másolás a kimenetre
  else if (vége a paraméterlistának)
        if (beépített)
        a megfelelő tevékenység végrehajtása
        else
        a definíció visszaküldése a bemenetre, miközben a $n-eket a paraméterek-
        kel helyettesítjük
        a verembeli terület ürítése
  else               # paramétermegőrzés
        a szimbólum átmásolása az aktuális veremterületre
```

Ez természetesen nem törődik olyan részletekkel, hogy a paraméterlista végén vagyunk-e és hogy milyen a verem szerkezete. Ezekre majd a továbbiakban térünk ki.

Mindenekelőtt nézzük a kiértékelő vermet! Ez nem más, mint az **evalst** nevű *(evaluation stack)* hosszú tömb. Az **evalst** első szabad helyének pozícióját **ep**-ban tároljuk, ami kezdetben 1. Valahányszor egy makrót dolgozunk fel, az **evalst** a nevet, a definíciót és a paramétereket leíró karaktersorozatokat tartalmazza. Az **argstk** tömb tartalmazza **evalst** azon címeit, ahol ezen jelsorozatok kezdődnek:

argstk(i) az *i*-edik **evalst**-beli jelsorozat kezdőpozíciója. Az **ap** az **argstk**-beli első szabad pozíciót tartalmazza; kezdetben ez is 1.

Mivel a makrók egymásba skatulyázhatók, az **evalst**-ban levő és az **argstk**-ban levő mutatók által megjelölt karaktersorozatok általában különböző szinten levő makrókhoz tartoznak. A verembeli területeket **callst** tartja számon: a **callst(i)** arra a pozícióra mutat **argstk**-ban, amely kijelöli az **i**-edik makró definíciójának szövegét. Az aktuális veremmutató **cp**. Ha **cp**=0, akkor nem vagyunk semmilyen makróban. Egyszintű makróhíváson belül a **cp**=1 és így tovább. Ennek megfelelően, ha egy harmadik szintű makróhívás első paraméterét keressük, először beállítjuk az **i=callst(3)** értéket. Ezután **argstk(i)** a makró definíciójának szövegére, **argstk (i+1)** a nevére, **argstk(i+2)** pedig az első paraméterére mutat.

A paraméterek összegyűjtése a zárójelek egyensúlyának vizsgálatát igényli, minden makrószinten egymástól függetlenül, ezért bevezettük a **callst**-vel párhuzamosan használatos **plev** tömböt, amely a megfelelő verembeli területhez tartozó zárójeleket számlálja.

A **cp**-t és a kimeneti puffert sok rutin használja, ezért ezeket a **cmacro** közös adatmezőben helyezzük el.

```
common /cmacro/ cp, ep, evalst(EVALSIZE)
    integer cp          # az aktuális hívás veremmutatója
    integer ep          # a következő szabad hely evalst-ban
    character evalst    # a kiértékelő verem
```

A többi változót — **callst**-ot, **argstk**-t, **ap**-t és **plev**-et — csak a főprogram használja, ezért ezeket nem tartjuk közös adatmezőben.

Van még egy utolsó probléma. A paraméterek összegyűjtésekor előkerülő makrókat azonnal kifejtjük. Van azonban olyan eset is, amikor a kiértékelést el kell halasztanunk. Tekintsük például a következő, arra irányuló kísérletet, hogy létrehozzunk egy új **d** makrót, amely a **define** szinonimája:

```
define(d,define($1,$2))
```

Felületesen szemlélve úgy tűnik, hogy ennek működnie kell, mivel a **d**-hez tartozó helyettesítő szöveg láthatóan **define($1, $2)**. A makrók és a beépített műveletek kiértékelése azonban előfordulásuk pillanatában történik. A belső **define** előbb kerül kiértékelésre, mint a külső. Mivel a **define**-hoz helyettesítő szöveg nem tartozik, végeredményben **d**-t üresnek definiáltuk, ami nyilván nem azonos azzal, amit el akartunk érni. A túl korai kiértékelés problémájának megoldása érdekében be kell vezetnünk egy „idézetképző" eljárást, mellyel a bemenetet, szükség esetén, változatlan jelsorozatként (literálisként) lehet kezelni. A mi megállapodásunk szerint a bemenet [és] közé zárt része változatlan marad, kivéve, hogy a zárójelpár eltávolítódik. Ennek segítségével a **d** makró a következőképpen írható fel:

```
define(d,[define($1,$2)])
```

Itt **d** helyettesítő szövegét a zárójelek védik, és ezért az változtatás nélkül **define ($1, $2)** lesz. Ezután már a

```
d (a, bc)
```

hívásnál minden jól működik és **a** definíciója **bc** lesz.

Akkor is szögletes zárójeleket kell használni, ha egy azonosítót újra definiálunk. A

```
define (x,y)
define (x,z)
```

definíciók eredménye **y**-nak a második sorban való definiálása, nem pedig **x** újradefiniálása. (Az első definíció még szintén érvényes, tehát **x** végül **y**-ná válik.) Ha nem akarjuk **y**-t újradefiniálni, a műveletsor a következő:

```
define (x,y)
define ([x],z)
```

A fenti példák külalakja nem tökéletes, mivel a vessző után nincsenek szóközök. A makrófeldolgozóban a szóközöknek jelentésük van és mi megpróbáltuk a felesleges szóközök létrehozását elkerülni.

A fenti megfontolások egyesítéséből egy elég hosszú, de valójában nem bonyolult főprogram adódik. A program a szögletes zárójelektől eltekintve az eddig kialakított vezérfonalat követi. A látszólag bonyolult szerkezet ellenére, egyszerűen egy hétágú **case** utasításról van szó, melyben az egyes esetek programjai nyílt rutinként a helyükön szerepelnek, nem alkotnak külön szubrutinokat. A főprogramot nem **define**-nak, hanem **macro**-nak neveztük el, mivel ez jobban kifejezi tevékenységét.

```
# macro — paraméteres makrók kifejtése
    character gettok
    character defn(MAXDEF), t, token(MAXTOK)
    integer lookup, push
    integer ap, argstk(ARGSIZE), callst(CALLSIZE), nlb,
        plev(CALLSIZE)
    include cmacro
    string balp "()"
    string defnam "define"
    integer deftyp(2)
    data deftyp(1) /DEFTYPE/, deftyp(2) /EOS/

    call instal(defnam, deftyp)
    cp = 0
    ap = 1
    ep = 1
    for (t=gettok(token,MAXTOK);t ¬= EOF;
            t=gettok(token, MAXTOK)) {
        if (t == ALPHA) {
            if (lookup(token, defn) == NO)
                call puttok(token)
            else {    # definiált; vigyük a kiértékelő verembe
                cp = cp + 1
                if (cp > CALLSIZE)
                    call error("a verem megtelt.")
                callst(cp) = ap
                ap = push(ep, argstk, ap)
                call puttok(defn)      # a definíció vermelése
                call putchr(EOS)
```

```
            ap = push(ep, argstk, ap)
            call puttok(token)      # a név vermelése
            call putchr(EOS)
            ap = push(ep, argstk, ap)
            t = gettok(token, MAXTOK)       # mi következik?
            call pbstr(token)
            if (t ¬= LPAREN)   # (és) beszúrása, ha hiányzik
                call pbstr(balp)
            plev(cp) = 0
            }
        }
else if (t == LBRACK) {    # a [ ] szint csökkentése eggyel
        nlb = 1
        repeat {
            t = gettok(token, MAXTOK)
            if (t == LBRACK)
                nlb = nlb + 1
            else if (t == RBRACK) {
                nlb = nlb − 1
                if (nlb == 0)
                        break
                }
            else if (t == EOF)
                call error("EOF a jelsorozatban.")
            call puttok(token)
            }
        }
else if (cp == 0)        # nem is vagyunk makróban
        call puttok(token)
else if (t == LPAREN) {
        if (plev(cp) > 0)
            call puttok(token)
        plev(cp) = plev(cp) + 1
        }
else if (t == RPAREN) {
        plev(cp) = plev(cp) − 1
        if (plev(cp) > 0)
            call puttok(token)
        else {                  # a paraméterlista vége
            call putchr(EOS)
            call eval(argstk, callst(cp), ap−1)
            ap = callst(cp)       # a kiértékelő verem ürítése
            ep = argstk(ap)
            cp = cp − 1
            }
        }
else if (t == COMMA & plev(cp) == 1) {    # új paraméter
        call putchr(EOS)
        ap = push(ep, argstk, ap)
        }
```

```
        else
            call puttok(token)      # csupán vermelni kell
        }
    if (cp ¬= 0)
        call error("váratlan EOF.")
    stop
    end
```

Fenn akarjuk tartani a **define** azon tulajdonságát, hogy egy paraméter nélküli makróhívás során (mint pl. **EOF** vagy **EOS**) ne kelljen zárójeleket használni. Ezért, ha egy szimbólum definiált név, és ezt nem követi egy bal oldali zárójel, visszaküldünk egy üres, kiegyensúlyozott zárójelpárt és így a paraméter nélküli makróhívásokat a későbbiekben nem kell majd speciális esetként kezelni. Ez újabb példa arra, hogy valamely adatábrázolás kisebb megváltoztatása árán megelőzhető a program nagyobb mértékű elbonyolódása.

Feltűnhetett az Olvasónak, hogy a szögletes zárójelek a makródefiníciókon kívül is törlődnek a szövegből. Bár ez a makrófeldolgozó felesleges mellékhatásának tűnik, nem ok nélkül választottuk ezt a megoldást. Legkézenfekvőbb példaként megemlíthetjük, hogy ha a „*define*" szót eredeti formájában akarjuk egy szövegben használni, meg kell védenünk egy zárójelpárral, ellenkező esetben a **define** beépített műveletként értelmeződik. Rövidesen más, lényegesebb indokokkal is találkozunk.

A **puttok** és a **putchr** jelsorozatokat, illetve egyes karaktereket küldenek rendre az **evalst**-ba (ha makrón belül vagyunk), vagy közvetlenül a kimenetre, a **putc** segítségével (ha makrón kívül vagyunk). Annak vizsgálata, hogy az átvitel melyik irányba történjék, egyetlen helyen dől el: a **putchr**-ben, és nincs szétszórva a program különböző helyein.

```
# puttok — egy szimbólum kivitele vagy vermelése
    subroutine puttok(str)
    character str(MAXTOK)
    integer i

    for (i = 1; str(i) ¬= EOS; i = i + 1)
        call putchr(str(i))
    return
    end

# putchr — egyetlen karakter kivitele vagy vermelése
    subroutine putchr(c)
    character c
    include cmacro

    if (cp == 0)
        call putc(c)
    else {
        if (ep > EVALSIZE)
            call error("kiértékelő verem megtelt.")
        evalst(ep) = c
        ep = ep + 1
        }
    return
    end
```

Amikor **evalst**-ba új paramétert kell elhelyezni, fel kell jegyeznünk az **ep** mutató aktuális értékét és meg kell növelni **ap**-t; ezt a **push** függvény végzi el:

```
# push — ep elhelyezése argstk-ban, új ap mutató visszaadása
   integer function push(ep, argstk, ap)
   integer ap, argstk(ARGSIZE), ep

   if (ap > ARGSIZE)
       call error("paraméterverem megtelt.")
   argstk(ap) = ep
   push = ap + 1
   return
   end
```

Amikor egy makró azonosítása befejeződött és összes paramétere összegyűlt az **evalst**-ban (ezt a zárójelszint nullára csökkenése jelzi), az **eval**-t kell behívni a beépített műveletek elvégzésére, illetve a megfelelő paraméterekkel ellátott definíciók visszaküldésére.

A **macro** a definíciót még a név előtt teszi az **evalst**-ba, ezért amikor **eval** hívásra kerül, **argstk(i)** a makró definíciójának szövegére, **argstk(i+1)** pedig a nevére mutat. A paraméterek mutatói **argstk(i+2)**-től **argstk(j)**-ig terjednek, számuk **j−i−1**. Ennek megfelelően **$0** maga a makrónév. Bár ezt valószínűleg nem nagyon fogják használni, a szabályosság mindenképpen elegáns.

```
# eval — i.—j.-ig terjedő paraméterek kifejtése a beépített függvények
#   kiértékelése, ill. a definíciók visszaküldése
   subroutine eval(argstk, i, j)
   integer index, length
   integer argno, argstk(ARGSIZE), i, j, k, m, n, t, td
   include cmacro
   string digits "0123456789"

   t = argstk(i)
   td = evalst(t)
   if (td == DEFTYPE)
       call dodef(argstk, i, j)
   else {
       for (k = t+length(evalst(t))−1; k > t; k = k − 1)
           if (evalst(k−1) ¬= ARGFLAG)
               call putbak(evalst(k))
           else {
               argno = index(digits, evalst(k)) − 1
               if (argno >= 0 & argno < j−i) {
                   n = i + argno + 1
                   m = argstk(n)
                   call pbstr(evalst(m))
                   }
               k = k − 1      # a $ átlépése
               }
```

```
        if (k == t)     # az utolsó karakter feldolgozása
            call putbak(evalst(k))
        }
    return
    end
```

Az **evalst**-ra való indexes hivatkozások legtöbbje tulajdonképpen résztömbökre vonatkozik, amit a 4. fejezetben tárgyaltunk. A PL/1 felhasználók jó, ha felfrissítik ezt a részt, mivel a fejezet sok programjában kell ezt a FORTRAN lehetőséget használni.

Ha a típus **define**, akkor **dodef**-et hívjuk; egyébként a definíciót visszaküldjük a bemenetre úgy, hogy minden **$***n*-et a megfelelő paraméterrel helyettesítünk. Az **ARGFLAG** szimbolikus konstans egy **$** jel, melyet természetesen a

```
define(ARGFLAG, DOLLAR)
```

paranccsal definiáltunk a **macro** RATFOR forrásprogramjában.

Még nem beszéltünk arról, hogy mi történjen a makrófeldolgozásban, ha egy makródefiníció nem létező paramétert kér. A legártalmatlanabb megoldás a paraméter kihagyása, ami gyakorlatilag a **$***n*-nek üres karaktersorozattal való helyettesítését jelenti — az **eval** pontosan ezt teszi. Ez akkor is igaz, ha nincs egyetlen paraméter sem. Ennek megfelelően, ha **x**-et a

```
define(x, a$1b)
```

definiálja, akkor az

```
x(+)
x(-, +)
x()
x
```

hívás mindegyike elfogadható eredményt ad, ami rendre: $a+b$, $a-b$, ab és ab.

A **dodef** egyszerű program, a feladat nagy részét már megoldották helyette:

```
# dodef — definíció bevitele a táblázatba
    subroutine dodef(argstk, i, j)
    integer a2, a3, argstk(ARGSIZE), i, j
    include cmacro

    if (j - i > 2) {
        a2 = argstk(i+2)
        a3 = argstk(i+3)
        call instal(evalst(a2), evalst(a3))     # résztömbök
        }
    return
    end
```

A makrófeldolgozó alkalmazásakor az első feladatok közé tartozik programozási nyelvünk szintaxisának bővítése. Bizonyos korlátozott mértékben mindvégig ezt tettük, amikor **character**-t írtunk, és **integer**-t értettünk rajta. A következő példa már bonyolultabb: A

```
define(proc, [integer function $1 $2 define(procname, $1)])
```

definíciónak megadott

```
proc(equal, (str1, str2))
```

bemeneti sor a következőt eredményezi:

```
integer function equal(str1, str2)
```

(A definíciót szögletes zárójelbe tettük, hogy elkerüljük a **define** túl korai kiértékelését. A definiáló szöveg zárójelbe tétele majdnem mindig hasznos.) A **proc** hívásának mellékhatásaként az **equal** függvénynév „megjegyzésre kerül", mivel a **procname**-t **equal**-ként definiáltuk. Ennek értékét más makrókban is felhasználhatjuk. Például, ha a **return**-t

```
define(return, [{procname=$1; [return] }])
```

formában definiáljuk, akkor írhatunk

```
return(YES)
```

alakú hívást, amely a FORTRAN **return** utasítását szimulálja és a PL/1-hez hasonlóan valamilyen értéket ad vissza. (A belső **return** körüli szögletes zárójelek megakadályozzák, hogy az a **return** makró újabb hívásaként értékelődjék ki; ez végtelen ciklust eredményezne. Ezt a példát célszerű gondosan tanulmányozni, ameddig nem válik világossá az egyes zárójelpárok szerepe.) Ha a **return** utasítással értéket tudunk visszaadni, programunk gyakran érthetőbbé válik, amint ezt az **equal** alábbi változatán is láthatjuk, ha azt összevetjük a 3. fejezetben szereplő változattal.

```
# equal — str1 és str2 összehasonlítása; az eredmény YES, ha egyenlők,
          NO, ha nem
   proc(equal, (str1, str2))
   character str1(ARB), str2(ARB)
   integer 1

   for (i = 1; str1(i) == str2(i); i = i + 1)
          if (str1(i) == EOS)
                 return(YES)
   return(NO)
   end
```

Gyakorlatok

8.12 Írjuk át a 3. fejezetben szereplő **filarg** függvényt a **return** makró felhasználásával! □

8.13 Ha a fent leírt **skipbl** makrót használjuk, gondoljunk egyet és inkább hívjunk helyette egy szubrutint. Át kell-e írni az összes hívást úgy, hogy tartalmazza a **call** szót? □

8.14 Írjuk meg azokat a makrókat, melyek a **getc**-t és a **putc**-t **getch**-ra és **putch**-ra való hivatkozásokra fordítják, ahogyan ezt a fejezet elején bemutattuk. Milyen problémák merülnek fel **getc**-vel kapcsolatban? □

8.15 A

define(sq,$1 * $1)

definíció egy kifejezést négyzetre emel. Tényleg ezt teszi? Mi lesz **sq(x+1)**-ből? Mit tudunk tenni az ügy érdekében? Mennyit kell tudnia egy makrófeldolgozónak a vele kapcsolatba kerülő nyelvek-ről? □

8.16 Dolgozzunk ki olyan szintaxist, amely lehetővé teszi, hogy egy makrónak 9-nél több paramétere legyen. Tegyük ezt kompatibilissé a **$***n* szintaxissal, ha $n<10$. Mennyire nehéz ezt megvalósítani? □

8.17 Tökéletesítsük a **define**-t úgy, hogy makródefiníció esetén lehetővé tegye formális nevek használatát a paraméterek megadására a **$***n* helyett. Azaz, ha *m* definíciója

define(m(x,y), az x és y szimbólumokat tartalmazó helyettesítő szöveg)

akkor az **m(a, b)** hívás **x** és **y** minden előfordulását rendre **a**-val és **b**-vel helyettesítse. Mennyit lehet a meglevő eszközökből felhasználni? □

8.6 FELTÉTELES MAKRÓK ÉS MÁS BEÉPÍTETT MŰVELETEK

A **macro**-t úgy terveztük, hogy könnyű legyen szükség esetén újabb beépített függvényeket hozzátenni. A fejlesztés következő állomása a feltételes makró bevezetése az **ifelse** beépített függvény segítségével. Az

ifelse(*a,b,c,d*)

hívás összehasonlítja az *a* és *b* jelsorozatokat. Ha azonosak, akkor a *c*-t, ha különbözőek, akkor pedig a *d*-t küldi vissza a bemenetre. Triviális példaként tekintsük a

define(compare, [ifelse($1,$2,yes,no)])

definíciót. A **compare** egy kétparaméteres makró, amely **yes**-t ad vissza, ha a paraméterek azonosak, és **no**-t, ha nem. Szokás szerint, a szögletes zárójelek megakadályozzák az **ifelse** idő előtti kiértékelését.

Ami ennél hasznosabb, tökéletesíthetjük a **return** makrót oly módon, hogy megvizsgálja, vannak-e paraméterei vagy nincsenek:

define(return, [ifelse($1,, [[return]], { procname=$1; [[return]] })])

Ha paraméterrel hívjuk, mint pl.

return (a+b)

esetén, **$1** nem üres és az eredmény:

{procname=a+b;return}

Ha nem volt paraméter, **$1** üres, ami az **ifelse** második paraméterével azonos és csupán egy egyedülálló **return** generálódik. Ez alkalommal kettős szögletes zárójelpár közé kellett tenni a generálandó **return** utasítást, kétszeres védelem céljából. Az első védelem az **ifelse** kifejtése közbeni kiértékelést akadályozza meg. A második védelemre — mint már említettük — azért van szükség, hogy a **return** utasítás a végső szövegben eredeti formájában szerepelhessen.

A beépített műveletek készletét két további művelettel bővítjük. Az egyik az **incr** *(increment)* növelő makró: Az

incr(x)

hívás számmá konvertálja az **x** jelsorozatot, ezt eggyel megnöveli, és ezen új szám jelsorozatként kifejezett alakját adja vissza helyettesítő szövegként. Célszerű, ha **x** numerikus karaktersorozat, különben nem kívánatos eredményt kaphatunk.

Az **incr**-t olyan feladat esetén használhatjuk, mint pl.:

```
define(MAXCARD, 80)
define(MAXLINE, [incr(MAXCARD)])
```

ahol két paramétert állítunk elő, melynek értéke 80, illetve 81. Ez akkor hasznos, ha két, egymástól kissé különböző két számot kell létrehoznunk, mint pl. az 1. fejezetben levő **getc**-ben és **putc**-ben. Két definíció, valamint annak a fejbentartása helyett, hogy ha az egyikben módosítunk, akkor a másikat is módosítani kell, egyszerűbb az egyik mennyiséget a másik segítségével definiálni. Az **incr** egy primitív aritmetikai képességet is nyújt bonyolultabb makrók létrehozására.

A másik beépített művelet a **substr** *(substring)*, amely egy jelsorozat részsorozatát választja ki. A

substr(*s,m,n*)

az *s* karaktersorozat azon részsorozatát állítja elő, amely az *m*-edik pozíciónál (kezdőpozíció: 1) kezdődik és hosszúsága *n*. Ha *n* nincs megadva vagy túl nagy, a sorozat teljes hátralevő része az eredmény, ha pedig *m* esik a karaktersorozaton kívülre, az eredmény az üres sorozat.
Például:

```
substr(abc,2,1)
```

eredménye **b**,

```
substr(abc,2)
```

eredménye **bc** és

```
substr(abc,4)
```

eredménye az üres sorozat.

Az **ifelse, incr** és **substr** létrehozása kis módosítást igényel. Kiegészítjük a **macro**-t az új kulcsszavak és a nekik megfelelő értékek beültetésével (ezek rendre: **IFTYPE, INCTYPE** és **SUBTYPE**, melyek mindegyike a többitől megkülönböztethető érték, majd módosítjuk **eval**-t, hogy ezeket is feldolgozza a **DEFTYPE**-hoz hasonlóan. Az **eval**-ba csak a vizsgálatokat és a szubrutinhívásokat kell beszúrni:

```
...
t = argstk(i)
td = evalst(t)
if (td == DEFTYPE)
    call dodef(argstk, i, j)
else if (td == INCTYPE)
    call doincr(argstk, i, j)
else if (td == SUBTYPE)
    call dosub(argstk, i, j)
else if (td == IFTYPE)
    call doif(argstk, i, j)
else {
    normális makrófeldolgozás az előzőek szerint
    }
...
```

A **doif** összehasonlítja első két paraméterét és a megfelelőt visszaküldi a bemenetre:

```
# doif — választás két paraméter közül
    subroutine doif(argstk, i, j)
    integer equal
    integer a2, a3, a4, a5, argstk(ARGSIZE), i, j
    include cmacro

    if (j - i < 5)
        return
    a2 = argstk(i+2)
    a3 = argstk(i+3)
    a4 = argstk(i+4)
    a5 = argstk(i+5)
    if (equal(evalst(a2), evalst(a3)) == YES)        # résztömbök
        call pbstr(evalst(a4))
    else
        call pbstr(evalst(a5))
    return
    end
```

A **doinc** konvertálja a számot, elvégzi az aritmetikai műveletet, majd az eredményt a **pbnum** segítségével jelsorozatként visszaküldi. Mivel a karakterek jobbról balra haladva keletkeznek, a sorrend megfordítására nincs szükség.

```
# doincr — a paraméter növelése eggyel
    subroutine doincr(argstk, i, j)
    integer ctoi
    integer argstk(ARGSIZE), i, j, k
    include cmacro

    k = argstk(i+2)
    call pbnum(ctoi(evalst, k)+1)
    return
    end

# pbnum — szám konvertálása jelsorozattá és visszaküldése az inputra
    subroutine pbnum(n)
    integer mod
    integer m, n, num
    string digits "0123456789"

    num = n
    repeat {
        m = mod(num, 10)
        call putbak(digits(m+1))
        num = num / 10
        } until (num == 0)
    return
    end
```

Végül a **substr** műveletet a **dosub** valósítja meg. Tevékenységének zömét az indexek megfelelő meghatározása teszi ki, különösen olyan szélsőséges esetekben, amikor a kívánt részsorozat valamilyen módon „kilóg" a sorozatból.

```
# dosub — részsorozat kiválasztása
    subroutine dosub(argstk, i, j)
    integer ctoi, length, max, min
    integer ap, argstk(ARGSIZE), fc, i, j, k, nc
    include cmacro

    if (j — i < 3)
        return
    if (j — i < 4)
        nc = MAXTOK
    else {
        k = argstk(i+4)
        nc = ctoi(evalst, k)          # karakterek száma
        }
    k = argstk(i+3)                   # kezdőpont
    ap = argstk(i+2)                  # a kívánt sorozat
    fc = ap + ctoi(evalst, k) — 1     # a részsorotat első karaktere
    if (fc >= ap & fc < ap + length(evalst(ap))) {   # résztömbök
        k = fc + min(nc, length(evalst(fc))) — 1
        for ( ; k >= fc; k = k — 1)
            call putbak(evalst(k))
        }
    return
    end
```

Gyakorlatok

8.18 Módosítsuk **doincr**-et úgy, hogy tetszőleges pontosságú aritmetikát végezzen! □

8.19 Valósítsuk meg az aritmetikát beépített műveletként: az

arith(első operandus, op, második operandus)

hívás végrehajtja az **op** által kijelölt műveletet a két (numerikus) operanduson. Elsőként a + és — műveleteket hozzuk létre. A szorzás, osztás, relációvizsgálat és a többi, szintén hasznos és egyszerűen megvalósítható. Mit kell módosítani a negatív számok helyes kezelése érdekében? Módosítsuk **arith**-ot tetszőleges pontosságú aritmetikájára! □

8.20 Definiáljunk egy **assert** makrót, melynek hatására egy programba feltételesen ellenőrző vizsgálatok fordíthatók. Ha az ellenőrzést bekapcsoljuk, az

```
assert(i < j)
```

hívás kifejtése nagyjából a következő:

```
if ( ¬ (i < j))
    call error("helytelen feltételezés...-ben: i < j")
```

ahol „. . ." az eljárás neve, melyet a **proc** makró őrzött meg. Valószínűleg olyan makróra is szükségünk lesz, amely a kívánt helyeken ki- és bekapcsolja az ellenőrzést. □

8.7 ALKALMAZÁSOK

Írjunk makrókat programjainkban használt karakterlánc-deklarációk egy változatának kezelésére. Tegyük fel, hogy a

```
string(name,"text")
```

a következőt rövidíti:

```
integer name(5)
data name(1) /LETt/
data name(2) /LETe/
data name(3) /LETx/
data name(4) /LETt/
data name(5) /EOS/
```

A feladat a **string** deklaráció kifejtése.

Szükségünk van a **text**-rész hosszára, ezért a **len** makróval kezdjük, amely meghatározza egy karakterlánc hosszát. Tehát a **len(abc)** makróhívás értéke 3, vagyis a paraméter hossza. Általánosan mi az *s* karakterlánc hossza? Ha *s* üres, akkor zérus; egyébként pedig a hosszúság eggyel több, mint azon részszorzat hossza, melyet *s*-ből egy karakter elhagyásával kapunk. Ez rekurzív definíció, amely a kifejezésnek természetes formája akkor, ha történetesen rekurzív a nyelvünk, mint jelen esetben is. Fejezzük ezt ki makrókkal:

```
define(len,[ifelse($1,,0,[incr(len(substr($1,2)))])])
```

Ez meglehetősen ijesztő, de a fenti rekurzív definíció alapján nem nehéz megérteni. Megengedett, és gyakran szükséges is, hogy makrókat saját magukkal definiáljunk. Ez azért lehetséges, mert a feltételvizsgálat felhasználható a végtelen ciklus megelőzésére. Jelen esetben a feltétel arra vonatkozik, hogy elhagytuk-e már a sorozat utolsó karakterét is.

A külső szögletes zárójelpár megakadályozza a kiértékelést a definíciónak a táblázatba való átmásolása alatt. A belső zárójelpár az **incr**-rész kiértékelését gátolja meg az **ifelse** paramétereinek összegyűjtése folyamán.

Ezután rátérünk magára a **string**-re. A feladat három részre oszlik: Először kiszámítjuk a pontos hosszúságot és a következőt generáljuk:

integer *név (hossz)*

Ezután egy ciklusban végigmegyünk a két idézőjel közötti karaktereken és mindegyikre a következő sort produkáljuk:

data *név* (*i*) /**LET** *c*/

ahol *c* a sorozat *i*-edik karaktere. A generálást végül a

data *név (hossz)* /**EOS**/

sorral zárjuk.

Maga a **string** makró a következő:

```
define(string,[integer $1(len(substr($2,2)))
str($1,substr($2,2),0)
data $1(len(substr($2,2)))/EOS/
```

A **len (substr($2,2))** makróhívás a karakterlánc tényleges hosszát határozza meg (az idézőjelek nélkül, de az **EOS**-sel együtt). A közbenső **data** utasításokat a **str** hozza létre:

```
define(str,[ifelse($2,",,data $1(incr($3))/[LET]substr($2,1,1)/
[str($1,substr($2,2),incr($3))])])
```

A **str** leválaszt egy karaktert, megnöveli az indexet, előállítja a sort és önmagát hívja rekurzíven, ameddig el nem éri a záró idézőjelet. (Miért van a **LET** szögletes zárójelek közé zárva?)

Láthatjuk, hogy ez a nyelv nem tartozik a legáttekinthetőbb programozási nyelvek közé. Hozzá kell szokni a használatához, mielőtt rekurzív ciklusokban gondolkoznánk, bár némi gyakorlattal jól elsajátítható. Ne akarjunk azonban túl okosak lenni a makrókkal. Elvileg a **macro**-val bármilyen számítástechnikai feladat megoldható, de nagyon könnyű meg nem érthető makrókat írni, melyek azután több bajt okoznak, mint amennyi munkát megtakarítottak.

Ezen kívül az is igaz, hogy a **string**-hez hasonló bonyolult rekurzív makróműveletek nagyon lassúak. Ennek illusztrálására két rövid — három-, illetve kilenckarakteres — sorozat feldolgozásának statisztikája következik:

	hívások száma	a központi egység ideje %-ban
gettok	2793	21,7
macro	1	11,6
puttok	1999	11,6
összes B/K	...	9,1
putchr	7700	7,5
ngetc	5977	6,4
type	5977	6,1
eval	211	5,4
lookup	648	5,0
putbak	5677	4,8
pbstr	431	3,4

Ilyen kevés adat feldolgozásához rengeteg szubrutinhívásra van szükség; ha kizárólag **string** makrókat dolgoznánk fel, a **macro** használhatatlannak bizonyulna. Szerencsére a **macro** igazi — nyelvi feldolgozók előtétjeként való — alkalmazása elsősorban egy karakterláncnak egy másikkal való helyettesítését igényli, amint ez a **define**-ban is történik. Ez sokkal kevésbé időigényes, ezért egy-egy **string** makró-feldolgozása gyakorlatilag is megengedhető. A makrónak az ilyen jellegű alkalmazások miatt megnövekedett bonyolultsága nem igényel sok többletidőt: azonos forrásszövegnél a **macro** mindössze 5%-kal fut tovább, mint a **define**.

A fenti mérések jól mutatják, hogy hová kell figyelmünket összpontosítani, ha a **macro**-t gyorsítani akarjuk. Az egyik lehetőség annak észrevétele, hogy egyes **gettok** hívásokat **ngetc** hívásokkal helyettesíthetünk, mivel egyetlen karakterről van szó (ilyen pl. egy szögletes zárójelben levő szöveg feldolgozása). Általánosabban fogalmazva, sok olyan apró rutin van, amelyeket a program megfelelő modularizálása érdekében írtunk. A **macro** idejének egy része a szubrutinhívó mechanizmus által elhasznált többletidő, ami egyes gépeken igen jelentős lehet. Ennek nagy részét kiküszöbölhetjük, ha a szubrutin hívásokat nyílt rutinokkal helyettesítjük (bár ezt mi a

szubrutintörzseket behelyettesítő makrók definiálásával és nem a programdarabok tényleges beszúrásával oldanánk meg). Pontosabban, mivel **putchr** majdnem minden hívása a **puttok**-ból ered, a **putchr**-t kis átalakítással be lehet építeni a **puttok**-ba. Ha a karakterek kis pozitív egész számok, a **type**-ot olyan hivatkozással helyettesíthetjük, amely egy — a megfelelő karakter típusát tartalmazó — tömbre vonatkozik. Ezt a hivatkozást közvetlenül a programban helyezhetjük el, ami lényegében nullára csökkenti a karaktertípusok felderítésének idejét. Ezen túlmenően, ha a **cdefio** közös adatmező a program más helyein is ismertté válik, az **ngetc**-ből és a **putback**-ből szintén lehet nyílt szubrutint készíteni.

Bár meg kell tartani egy program viszonylagos áttekinthetőségét, a hatékonyság elérhető javulása számottevő lehet. A **macro** eredeti változatát, melyet *C* nyelven írtunk, ilyen módosításokkal mintegy négyszeresére gyorsítottuk fel. A legtöbb gépen, FORTRAN-ban hasonló eredmény várható. A folyamat azonban — mint ezt már többször említettük — a következő kell, hogy legyen: Írjunk egy áttekinthető programot a megfelelő algoritmusra, mérésekkel határozzuk meg a szűk keresztmetszeteket, majd gyorsítsuk a programot ezeken a helyeken, a lehető legáttekinthetőbb módon. Ha a dolgot a másik végén fogjuk meg, szinte biztos, hogy áttekinthetetlen zagyvasághoz jutunk.

A makrók gyorsabbá és hatásosabbá tételének egyik módja a beépített műveletek készletének bővítése, hogy a **len**-hez hasonló számításokat ne kelljen teljes részletességgel leírni. Néhány ötletünket a most következő gyakorlatok tartalmazzák.

Gyakorlatok

8.21 Alakítsuk át a **len**-t beépített műveletté! □

8.22 Valósítsuk meg az **index** beépített műveletet, a 2. fejezetben szereplő **index** függvény mintájára: **index(s,c)** legyen a **c** karakternek az **s** karakterláncon belüli pozíciója, és zérus, ha **c** nem fordul elő **s**-ben. Meg tudjuk-e valósítani **index**-et a meglevő lehetőségeinkkel? Célszerű-e ezt megtennünk? □

8.23 A **string** fenti megvalósítása csak betűkből és számokból álló karakterláncok feldolgozására alkalmas. Általánosítsuk úgy, hogy nem alfanumerikus jeleket tartalmazó sorozatokat is kezelni tudjon! □

8.24 Írjuk meg az „**err**" makrót, amely az

err ("üzenet")

alakú hívásokat a tiszta FORTRAN-beli

call error (7, 7hüzenet)

utasításokká alakítja. Az üzenetet bővítsük ki a hívó rutin nevével. □

8.25 Írjuk meg a **count** makrót, amely megszámolja egy szubrutin hívásainak számát oly módon, hogy a rutinba való minden belépéskor egy előre definiált outputfile-ba beírjuk a rutin nevét. Milyen más, már kifejlesztett eszközt használnánk egy futás eredményeinek összefoglalására? □

8.26 Milyen módosításokat kellene a **macro**-n végrehajtani ahhoz, hogy alkalmas legyen a 7. fejezetben szereplő **format** program számára megfelelő makrók kezelésére? □

BIBLIOGRÁFIAI MEGJEGYZÉSEK

A makrófeldolgozás irodalma sokkal nagyobb annál, semhogy az itt rendelkezésre álló helyen bemutathatnánk. Az M. Campbell—Kelly által írott *„An Introduction to Macros"* (American-Elsevier, 1973) röviden tárgyalja a makrófeldolgozók számos különböző fajtáját. A P. J. Brown-tól származó *„Macro Processors and Techniques for Portable Software"* (Wiley 1974) c. mű részletesebben foglalkozik a makrófeldolgozás alapvető szempontjaival.

A PL/1 makró-előfeldolgozó megkísérel egy olyan makrónyelvet adni, amely lényegében azonos a fordítóprogram nyelvével. Ezt több PL/1-gyel foglalkozó anyag, valamint az egyes gépi reprezentációk hivatkozási kézikönyvei tárgyalják. Lásd pl. *IBM System*/360, *PL/1 Language Specification,* Form Y33—6003; vagy *Student Text: An Introduction to the Compile-Time Facilities of PL/1,* Form C20—1689.

A makrók igen hasznosak „hordozható" software fejlesztésekor, azaz olyan programok esetén, melyeket egyik gépről a másikra a teljes átírásnál lényegesen kisebb munkával lehet átvinni. A programot egy szerény mennyiségű makrót tartalmazó készletre alapozva írják meg. Egy-egy adott környezet esetén csakis a makrókat kell megírni. Valószínűleg a SNOBOL az így megvalósított fontosabb nyelvek ismertebb képviselője. Lásd: R. E. Griswold, J. F. Poage and I. P. Polonsky: *„The SNOBOL 4 Programming Language"* (Prentice-Hall, 1969) vagy R. E. Griswold: *„The Macro Implementation of SNOBOL 4"* (Freeman, 1972). Brown könyve ezzel a témával foglalkozó egyéb tevékenységeket tárgyal.

Minden adatszerkezetekkel foglalkozó könyv tárgyalja a táblázatos formájú információ kezelésével kapcsolatos problémákat. A szokás szerinti irodalmi hivatkozás D. E. Knuth: *„The Art of Computer Programming"* (Addison-Wesley) c. könyvének 1. kötete (1968), amely keresési technikákkal foglalkozik igen részletesen.

A fejezetben leírt makrófeldolgozót eredetileg D. M. Ritchie tervezte és programozta a *C* nyelven. Köszönetünket fejezzük ki, hogy rendelkezésünkre bocsátotta.

9. fejezet

A RATFOR—FORTRAN fordító

A könyvben szereplő programokat RATFOR-ban mutatjuk be. Most rátérünk annak tárgyalására, hogyan lehet olyan programot írni, amely a RATFOR-t FORTRAN-ra fordítja. A fordító bemutatásának egyik célja, hogy részletesen ismertessünk egy általunk rendszeresen használt eszközt, amelynek szerintünk minden FORTRAN felhasználó számára gyakorlati értéke is van. Amint ezt a Bevezetésben említettük, a FORTRAN egyáltalán nem halott nyelv; használata széles körű és értékes tulajdonságai vannak, amelyek ellensúlyozzák gyenge pontjait. Ha valaki FORTRAN-t használ, használja azt minél hatékonyabb módon.

A fejezet másik célja egy komolyabb méretű előfeldolgozó bemutatása. Az előfeldolgozó azért előnyös, mert új fordítóprogram írása nélkül, mások munkájára támaszkodva juthatunk egy jobb programozási nyelvhez. Ha az Olvasónak nincs is szüksége a RATFOR-ra mint olyanra, lényeges annak belátása, hogy gyakran lehet kényelmes, illeszkedési felületet teremteni egy szoftver termékkel vagy hiányzó szolgáltatásokat pótolni viszonylag szerény közbenső programok létrehozásával. A RATFOR tervezése és megvalósítása hasznos ötleteket adhat hasonló eszközök létrehozására.

A RATFOR elsődleges célja a FORTRAN nyelv javítása volt — mind programozási, mind publikációs célból — oly módon, hogy lehetőséget adjon és bátorítsa az Olvasót jól strukturált programok írására. Ezt a célt az eredeti FORTRAN-ból hiányzó vezérlési szerkezetek bevezetésével és a nyelv „kozmetikázásával" valósítja meg.

A vezérlési szerkezetek, amelyeket tárgyalunk, az **if-else, while, do, break, next,** valamint a kapcsos zárójelekkel csoportosított utasítások. A **for**-t és a **repeat-until**-t az előzőekből következő gyakorlatokként bízzuk az Olvasóra. Ezek a szerkezetek teljesen és kényelmesen elegendőek a **goto** nélküli programozáshoz. Bár a kérdéssel kapcsolatban nincsenek „vallási meggyőződéseink", feltűnhetett az Olvasónak, hogy RATFOR programjainkból hiányzik a **goto**. Ez az elv nem jelentett számunkra korlátozást, mivel egy megfelelően kifejező erejű nyelv és egy kicsit gondos programozás esetén a **goto**-ra csak igen ritkán van szükség.

A RATFOR „csinosításával" a tömörséget és az elfogadhatóan tetszetős külalakot igyekeznek megvalósítani. A nyelv szabad formátumú: az utasítások a forrássorban bárhol állhatnak. A sor vége általában egy utasítás végét is jelzi, de láthatóan befejezetlen utasítások, mint pl. az **if** és **while** utasításokon belüli összetett feltételek vagy vesszőre végződő sorok automatikusan folytatódnak a következő sorban. Egy soron belül, pontosvesszővel elválasztva, több parancs is állhat, bár ennek gyakorlati alkalmazását nem ajánljuk. A megjegyzések írására vonatkozó szabály — egy # számjel a sorban bárhol a megjegyzés kezdetét jelzi — fontos, tömör magyarázatok írására ösztönöz. Az idézetek a régimódi FORTRAN fordítók számára *n***h** formátum-

ra konvertálódnak, így a programozónak nem kell karaktereket számlálnia. A > jelhez hasonló jelölések gyorsabban közvetítik a program jelentését, mint a vele azonos FORTRAN szimbólumok, mint pl. a **.gt.**.

Természetesen a FORTRAN-nak vannak gyenge pontjai a vezérlési szerkezeteken és a külalakon kívül is. Amint ez programjaink némelyikében látható, a rendelkezésre álló (vagy nem álló) adatszerkezetek komoly mértékben bonyolíthatják a programozást. A mi előfeldolgozónk nem próbálja felszámolni ezeket a hiányosságokat, bár egy bonyolultabb változat képes lehetne erre is. A RATFOR nem kíván teljesen új nyelv lenni, csupán segíteni kíván a FORTRAN legkomolyabb hiányosságainak kiküszöbölésében, hogy ezáltal megfelelő publikációs és programozási nyelvvé tegye azt.

Sok, széleskörűen elterjedt FORTRAN előfeldolgozó létezik. Bár ezek látszatra sok mindenben különböznek, legtöbbjük a RATFOR-hoz hasonló szolgáltatásokat nyújt. Ha valamelyik az Olvasó rendelkezésére áll, mindenképpen érdemes használni — bármely előfeldolgozás komoly előnyöket biztosít.

9.1 SZERVEZÉSI KÉRDÉSEK

Egy programozási nyelv leírásának egyik kényelmes módja a Backus-Naur Forma (BNF), amely a nyelvhez tartozó nyelvtan formális specifikációja, azaz azon szabályok halmaza, melyek segítségével a nyelv szerint hibátlan program írható vagy olvasható. Sok előny származik abból, hogy a nyelvet egy nyelvtan és nem kötetlen szöveges leírás segítségével határozunk meg. Ilyen módon a nyelv specifikációja igen pontos lehet, elkerülhetők az olyan pongyolaságok és kétértelműségek, melyek egy természetes nyelvű leírásban valószínűleg előfordulnának. Továbbá egy program, melyet „compiler—compiler”-nek hívnak, fel tud használni egy adott nyelvtant arra, hogy olyan programot hozzon létre, amely ezen a nyelven írt programok szintaktikus elemzésére alkalmas. Nagy és bonyolult nyelvek esetén ez az automatizálás igen értékes a megbízható és könnyen módosítható elemzők létrehozásának szempontjából. Mivel az elemző a fordítóprogram központi része, a compiler—compiler jobb minőségű fordítóprogram kidolgozásához is vezet.

Szerencsére, a RATFOR nyelvtana elég szűkös és egyszerű, semhogy egy compiler—compiler-t igényeljen, jóllehet eredetileg ennek segítségével valósították meg a nyelvet. Mindenesetre a RATFOR-t a következő nyelvtannal definiáljuk:

program: *utasítás*

| *program utasítás*

utasítás: **if** *(feltétel) utasítás*

| **if** *(feltétel) utasítás* **else** *utasítás*

| **while** *(feltétel) utasítás*

| **for** *(kezdőérték; feltétel; léptetés) utasítás*

| **repeat** *utasítás*

| **repeat** *utasítás* **until** *(feltétel)*

| **do** *határok utasítás*

| *számjegyek utasítás*

| **break**

| **next**

| { *program* }

| **other**

Az első két sor azt jelenti, hogy egy *program* vagy *utasítás*, vagy *program*, melyet egy *utasítás* követ. Más szavakkal, egy program egy vagy több utasításból áll. Az utasítás pedig meglehetősen sokféle lehet: a | függőleges vonal a különböző változatok közüli választási lehetőségeket jelöli. A legtöbb utasítás egyszerű szerkezetű, a megfelelő kulcsszóval jellemezhető. Például egy utasítás állhat az **if** kulcsszóból, egy zárójelek közé tett *feltételből* és egy *utasításból*. (A definíció rekurzív, akárcsak a program definíciója.) Egy kapcsos zárójelek között álló utasításcsoport bárhol használható, ahol utasítás megengedett. Ez az

utasítás: { *program* }

szabályból következik. A kevésbé mesterkélt kapcsos zárójelet használjuk a jobban elterjedt **begin—end**, vagy **do—end** helyett. Bekezdések segítségével a program szerkezete jobban kifejezhető, mint nagybetűs kulcsszavakkal. Ha azonban valakinek nagyon szűkös jelkészlete van, más megoldást is választhat, a gyakorlatokban szerepelnek erre vonatkozó javaslatok.

A *„számjegyek"* fogalom egy számjegysorozatot jelent, azaz a szabványos FORTRAN utasítás számot vagy címkét. Bár ez RATFOR-ban viszonylag ritka, a nyelvtannak tartalmaznia kell a definíciót.

A **next** olyan utasítás, amely egyik programban sem fordult elő, pedig más alkalmazásoknál hasznos. Rokon a **break**-kel, de a ciklusból való kilépés helyett a ciklus következő iterációjának elindítását eredményezi. A **while, repeat** vagy **do** ciklusban a **next** hatására a vezérlés közvetlenül a feltétel részre adódik át, a **for** esetén pedig a léptetésre.

Az utolsó nyelvtani típus az **other;** ide tartozik minden, amit a megelőző típusok egyikeként sem ismertünk fel. Lényegében ez a fogalom tartalmazza a FORTRAN legnagyobb részét. Például az

i = 1

utasítás nem **if, else** vagy más, a RATFOR által felismerhető szerkezet, ezért az **other** típushoz tartozik.

Az **other** típus bevezetése lényeges egyszerűsítést jelent, mivel lehetővé teszi, hogy a RATFOR ne tudjon sokat a FORTRAN-ról. Ha olyan utasítás kerül elő, amely nem a kulcsszavak egyikével (illetve számjegyekkel vagy bal oldali kapcsos zárójellel) kezdődik, akkor *szükségképpen* **other** típusú, és igazi feldolgozást nem igényel. Az egyszerűsítés ára az, hogy a RATFOR hibafelderítő képessége nem olyan jó, mint egy részletesebb nyelvtan esetén lehetne.

Elvileg a nyelvtan minden szabályához egy *szemantikai tevékenység* tartozik, amely meghatározza, hogy mi a teendő, amikor az adott szerkezetet a fordítás közben a programban felismertük. A RATFOR esetén a szemantikai tevékenységek rendszerint nagyon egyszerűek, mivel csupán a bejövő szöveg formai módosítására, valamint a vezérlési szerkezetek fordításához szükséges **if**-ek, **goto**-k és **continue** utasítások beszúrására vonatkoznak.

Az előfeldolgozó szerkezete a következő: A legfelső szint egy vezérlő rutin, melyet *„elemző"*-nek hívnak, mivel a forrásszöveg nyelvtani szerkezetének elemzése útján vezérli a fordítást. Például, ha egy **if** érkezik, az elemző meghívja azt a rutint, amely az **if** utasítást dolgozza fel. Ez a rutin leválasztja a *feltétel* részt és egy szemantikai tevékenység keretében generálja a vizsgálatot. Az elemzőnek meg kell jegyeznie, hogy **if** utasítást látott, hogy az „utasítás" rész végén az **if**-re vonatkozó „utasítás vége" kódot állíthassa elő. Ezen tevékenységhez az **else** feldolgozása is csatlakozhat,

ha az utasítás **else**-t tartalmaz. Továbbá, az egész elemzés eredendő módon rekurzív, amint ezt a BNF alak mutatja, mivel az egyes szerkezetek egymásba skatulyázhatók, mint pl. a következő utasítások:

```
for (i = 1; i <= n; i = i + 1)
     for (j = 1; j <= n; j = j + 1)
          if (m(i, j) < 0)
               m(i, j) = - 1
```

Az elemző minden utasítás elején behívja a „lexikális analízis" nevű rutint, amely az utasítást a nyelvtanban szereplő valamelyik típusba sorolja. A lexikális elemző rutin egy alacsonyabb szintű rutint hív az utasítás első szimbólumának kiválasztására, amely meghatározza az utasítás típusát. Mikor az utasítás típusa ismert, az elemző behívja a megfelelő kódgeneráló rutint. Egyes kódgeneráló rutinok az utasítás később következő részeinek feldolgozása során szintén használják a szimbólumolvasó rutint.

Először a lexikális és a szimbólumolvasó rutint tárgyaljuk, mivel ezek lényegében minden mástól függetlenek. Ezután egyszerűbb lesz az elemző és a kódgeneráló bemutatása.

9.2 LEXIKÁLIS ANALÍZIS

A RATFOR szimbólumai hasonlóak a **define** és a **macro** szimbólumaihoz, melyek a 8. fejezetben szerepeltek; mindössze az idézeteket kell hozzávenni az utóbbihoz. A **gettok** a bemenetet alfanumerikus jelekre, idézetekre, illetve egyedülálló, nem alfanumerikus jelekre bontja. Ugyanakkor a szimbólumokat elválasztó szóközöket, tabulátorokat és megjegyzéseket elhagyja. (A szóközök elhagyhatók, mivel FORTRAN programokban nincs jelentőségük. Ez egyike azon számos esetnek, amikor a FORTRAN programokban megengedett nagyfokú szabadság a kezünkre játszik, és egyszerűbbé teszi a RATFOR feladatát.) A **gettok ALPHA**-t ad vissza, ha alfanumerikus karakterláncot talált, különben magát a nem alfanumerikus karaktert kapjuk.

```
# gettok — egy szimbólum előállítása a RATFOR számára
   character function gettok(lexstr, toksiz)
   character ngetc, type
   integer i, toksiz
   character c, lexstr(toksiz)
   include cline

   while (ngetc(c) ¬= EOF)
        if (c ¬= BLANK & c ¬= TAB)
             break
   call putbak(c)
   for (i = 1; i < toksiz−1; i = i + 1) {
        gettok = type(ngetc(lexstr(i)))
        if (gettok ¬= LETTER & gettok ¬= DIGIT)
             break
        }
```

```
if (i >= toksiz-1)
    call synerr("túl hosszú szimbólum.")
if (i > 1) {                       # alfanumerikus jel érkezett
    call putbak(lexstr(i))         # eggyel túlmentünk
    lexstr(i) = EOS
    gettok = ALPHA
    }
else if (lexstr(1) == SQUOTE | lexstr(1) == DQUOTE) {
    for (i = 2; ngetc(lexstr(i)) ¬= lexstr(1); i = i + 1)
        if (lexstr(i) == NEWLINE | i >= toksiz-1) {
            call synerr("idézőjel hiányzik.")
            lexstr(i) = lexstr(1)
            call putbak(NEWLINE)
            break
            }
    }
else if (lexstr(1) == SHARP) {     # megjegyzések kihagyása
    while (ngetc(lexstr(1)) ¬= NEWLINE)
        ;
    gettok = NEWLINE
    }
lexstr(i+1) = EOS
if (lexstr(1) == NEWLINE)
    linect = linect + 1
return
end
```

A **gettok** ugyanazokat a B/K visszaküldő rutinokat használja, melyeket a 8. fejezetben tárgyaltunk a **define**-nal, illetve a **macro**-val kapcsolatban: a **putbak** egyetlen karaktert küld vissza a bemenetre; az **ngetc** egy karaktert olvas, ami visszaküldött karakter is lehet. A **type** a 4. fejezetből származik, és **LETTER**-t vagy **DIGIT**-et ad vissza aszerint, hogy paramétere betű vagy számjegy, illetve nem numerikus paraméter esetén magát a karaktert adja vissza.

Az idézetek biztonsági okokból nem léphetik át a sorhatárt. Az ilyen karakterláncban előforduló sorvége majdnem mindig egy hiányzó idézőjelet jelent, ami — ha nem vesszük észre — az egész programot tönkreteheti. Ha az idézőjelek között **NEWLINE** is előfordulhatna, a **gettok** egyszerűbb lehetne, de elvesztenénk egy értékes hibaellenőrzési lehetőséget, a program többi részének pedig sokkal gondosabban kellene ellenőriznie *saját adatait*. Legjobb a hibalehetőséget minél korábban kiküszöbölni.

Mivel a **gettok** szóközöket használ a szimbólumok elválasztására, a szóközöknek a RATFOR-ban van jelentésük, a FORTRAN-ban viszont nincs. A kulcsszavak (pl. az **if**) nem tartalmazhatnak szóközt, mivel ekkor nem tudjuk őket felismerni. Bár a FORTRAN lényegében mindenhol figyelmen kívül hagyja a szóközöket, ezzel a szabadsággal inkább visszaélnek, mintsem ésszerűen kihasználnák. Mi természetesen jól megvagyunk a szóközöket tartalmazható RATFOR kulcsszavakból adódó bonyodalmak nélkül is.

A **synerr**-t sok helyről hívják szintaktikus hibaüzenetek kinyomtatása céljából, így a **gettok**-ból is, két helyről. A **gettok** és a **synerr** egy közös adatmezőbeli változón: a forrássorok számlálóján, a **linect**-en osztozik. Így a hibás sorokat a forrás-

szövegen belüli pozíciójukkal lehet megadni. A **linect**-et a **cline** nevű közös adatmezőn keresztül adjuk át. Bár ezt csupán a **gettok** és a **synerr** használja, mégis közös adatmező kell, hogy legyen, különben majdnem minden rutinnak tartalmaznia kellene a sorszámot, hogy az szükség esetén rendelkezésre álljon. Ez olyan eset, amikor egy rejtett adatkapcsolatot ténylegesen arra használunk, hogy csökkentsük a programrészek általános összefonódását.

```
common/cline/ linect
integer linect     # az inputfile sorszámlálója; kezdetben 1
```

A **synerr** kódja a következő:

```
# synerr — RATFOR szintaxisbeli hiba jelzése
   subroutine synerr(msg)
   character lc(MAXLINE), msg(MAXLINE)
   integer itoc
   integer junk
   include cline

   call remark("a hibás sor száma.")
   junk = itoc(linect, lc, MAXLINE)
   call putlin(lc, ERROUT)
   call putch(COLON, ERROUT)
   call remark(msg)
   return
   end
```

Az **itoc** a 2. fejezetből való, a sorszámot nyomtatásra alkalmas jelsorozattá konvertálja.

Míg a FORTRAN lexikális elemzése bonyolult, a RATFOR-é egyszerű. Az azonosításra kerülő típusok csupán literális formában adott karakterek (mint pl. a pontosvessző vagy a kapcsos zárójel), kulcsszavak (mint pl. **if** és **else**) címkék, melyek csupa számjegyből állnak, vagy fel nem ismert szimbólumok lehetnek. Ezt az azonosítást végzi el a **lex** rutin:

```
# lex — szimbólum lexikális típusának meghatározása
   integer function lex(lexstr)
   character gettok
   character lexstr(MAXTOK)
   integer alldig, lookup
   integer ltype(2)

   while (gettok(lexstr, MAXTOK) == NEWLINE)
         ;
   lex = lexstr(1)
   if(lex==EOF|lex==SEMICOL|lex==LBRACE|lex==RBRACE)
         return
   if (alldig(lexstr) == YES)
         lex = LEXDIGITS
```

```
    else if (lookup(lexstr, ltype) == YES)
        lex = ltype(1)
    else
        lex = LEXOTHER
    return
    end
```

Először elhagyjuk az üres sorokat. Ha az első szimbólum pontosvessző, kapcsos zárójel vagy **EOF**, ez lexikális értékként adódik vissza. Ha a szimbólum csak számjegyeket tartalmaz, akkor szükségképpen egy FORTRAN címke, mivel egy utasítás első szimbólumáról van szó. RATFOR programokban címkék általában csak **format** utasítások előtt jelennek meg, bár tetszőleges utasítás előtt állhatnának. Címke esetén a visszaadott lexikális érték **LEXDIGITS**.

Ha a szimbólum nem címke, meg kell vizsgálni, hogy benne van-e a kulcsszavak (**if, else** stb.) táblázatában. Ha igen, **lex** a megfelelő kulcsszótípust adja vissza, különben a visszaadott érték **LEXOTHER**. A táblázatban való keresést a 8. fejezetben használt **lookup** rutin végzi. Az elemző állítja be a szimbólumtábla kezdőértékeként a kulcsszavak neveit és a **lex** által visszaadandó típusértékeket. A típusok egyelemű jelsorozatok, az értékek pedig a **LEXIF, LEXDO** stb. szimbolikus konstansok. Az eredmény **ltype**-ban adódik vissza (amely szükségképpen kételemű tömb, a típus és az **EOS** tárolására). A típusoknak különbözniük kell az **EOF**-től, a pontosvesszőtől és a kapcsos zárójelektől — mivel a **lex**-től ezeket szintén kaphatjuk eredményként — más megszorítás azonban nincs.

Az **alldig** eldönti egy jelsorozatról, hogy az csupa számjegyből áll-e.

```
# alldig — az eredmény YES, ha str csupa számjegyből áll
    integer function alldig(str)
    character type
    character str(ARB)
    integer i

    alldig = NO
    if (str(1) == EOS)
        return
    for (i = 1; str(i) ¬= EOS; i = i + 1)
        if (type(str(i)) ¬= DIGIT)
            return
    alldig = YES
    return
    end
```

9.3 KÓDGENERÁLÁSI SZABÁLYOK

Az 1. fejezetben röviden utaltunk arra, hogy hogyan lehet a RATFOR vezérlő utasításait mechanikusan FORTRAN-ra fordítani. Az ott szereplő hozzávetőleges leírást most pontosítjuk.

if *utasítás:*

Az
if *(feltétel) utasítás*
fordítása lényegében
if (*a feltétel* nem igaz) kerüld ki az *utasítást* alakú. Ezért ha egy **if** érkezik, a következő feladataink vannak:
— a *feltétel* rész leválasztása
— egy **L** egyedi címke előállítása és tárolása
— az „**if (.not.** *(feltétel)*) **goto L**" utasítás kivitele.
(FORTRAN-ban a **.not.***(feltétel)* szerkezet invertálja a *feltétel* logikai értékét.) Amikor az **if**-et követő utasítás végéhez érünk, két lehetőség van. Ha nincs **else** az utasításban, egyszerűen ki kell vinni az

```
L continue
```

utasítást. Ha azonban itt **else** áll, egy másik, **L1** címkét kell generálni, és kivinni a

```
  goto L1
L continue
```

utasításokat az **else** ág átugrására, majd az **else**-t követő bármilyen utasítás után az

```
L1 continue
```

sort az **if-else** szerkezet lezárása céljából. Összefoglalva, az

if *(feltétel) utasítás*

esetén generált kód a következő:

```
  if (.not. (feltétel)) goto L
            utasítás
L continue
```

Az
if *(feltétel) utasítás1* **else** *utasítás2*
utasítás esetén generált kód pedig

```
  if (.not. (feltétel)) goto L
            utasítás1
            goto L1
L continue
            utasítás2
L1 continue
```

Mivel a címkék „ingyen vannak", abban az értelemben, hogy a szabályos utasításszámok egyik programban sem fogynak el, legegyszerűbb minden **if** esetén két, egymást követő címkét generálni: **L1=L+1**. Ha egyikükre nincs szükség, mert az **else** nincs az utasításban, nem vesztettünk semmit. Mivel pedig tudjuk, hogy a címkék mindig **L** és **L+1** alakúak, elég az egyiket tárolni, a másikat 1 hozzáadásával kiszámíthatjuk.

do *utasítás:*

A RATFOR **do** utasítása egy címke nélküli FORTRAN **do.**
Előfordulásakor teendőink a következők:
— a határok leválasztása,

— az **L** címke előállítása,

— a „**do L** *határok*" utasításkivitele.

A **do**-hoz tartozó utasítás végén pedig ki kell vinni az

L continue

sort. Ily módon a RATFOR

do *határok utasítás*

alakú utasításának fordítása:

```
do L határok
     utasítás
L    continue
L+1  continue
```

A második **continue** akkor szükséges, ha a ciklusban egy **break** utasítás fordul elő: a **break** a

goto L+1

míg a **next** utasítás a

goto L

sort generálja.

Mi a **break** jelenlététől függetlenül mindig generáljuk a második **continue**-t, mivel a **break** jelenlétének vagy jelen nem létének vizsgálata túl sokba kerülne. A kódgeneráló feladatának egyszerűsítésekor azt a tényt használjuk ki, hogy a FORTRAN fordítók rendszerint ésszerűen kezelik a felesleges **continue** utasításokat.

while *utasítás:*

A **while** utasítás az **if** és a **do** tulajdonságainak egyszerű kombinációja. A

while *(feltétel) utasítás*

fordítása:

```
     continue
L    if(.not. (feltétel)) goto L+1
          utasítás
          goto L
L+1  continue
```

L+1 a **break** címkéjeként is felhasználható, a **next** egyszerűen **L**-re adja a vezérlést. Az **if** előtti **continue**-ra a valószínűtlen, de elképzelhető címkézett **while** kedvéért van szükség, pl. a

10 while (i > 0)...

utasítás esetében. Ekkor a RATFOR a **10**-es címkét, a **while** számára generált programrész első utasításának, vagyis a **continue**-nak az utasításszám-mezőjében helyezi el. A RATFOR-ban ritkán válik szükségessé utasításszámok használata, de a jó tervezés alapelve, hogy minden értelmes lehetőséget megengedjünk.

Amikor címkét (lexikális típusa: **LEXDIGITS**) találunk, azt kivisszük a sor első pozíciójától kezdve, annyi szóközzel kiegészítve, hogy a következő karakter a 7. oszlopba kerüljön, ami egy FORTRAN utasítás szabványos kezdő pozíciója. A **LEXOTHER** típusú input-ot csak át kell másolni a kimenetre annyi formázással, hogy az megfeleljen a FORTRAN követelményeinek; a szöveg rendszerint a sor 7. és 72. pozíciója között helyezkedik el.

Látható, hogy a RATFOR minden utasításhoz a kézenfekvő kódot generálja, és nem is próbálkozik speciális esetek optimalizálásával. Erre jó példa az **if** utasítás. Az

```
    if (feltétel) utasítás
```

(**else** nélkül) számára generált kód *mindig*

```
    if (.not. (feltétel)) goto L
        utasítás
  L continue
```

alakú még akkor is, ha az **if** utasítás a FORTRAN-ban is megengedett formájú, azaz le sem kellett volna fordítani, mint pl. az

```
  if (feltétel) return
```

esetén.

Ez a megoldás azért indokolt, mert a legtöbb esetben az „optimalizálás" elhanyagolható hatású. Az általunk használt több különböző fordítóprogram esetén a többlet általában csak egy felesleges **goto** utasítás, mely a futási időt alig kimutatható mértékben befolyásolja. Ritkán fordul elő, hogy a generált kód átgyúrásával számottevő javítás érhető el. (Végezzünk méréseket, mielőtt bonyolultabbá tesszük a kódgenerálást!) Az algoritmuson, illetve az adatszerkezeten eszközölt változtatásokkal hatásosabban javíthatjuk a teljesítményt.

Nem hagyhatjuk említés nélkül, hogy ennek az egyszerű kódgenerálásnak a hátránya, hogy egyes szerkezetek hibaüzenetet válthatnak ki a FORTRAN fordítóból. Az

```
  if (feltétel)
      return
  else
      a = b
```

utasításokból generált kód:

```
      if (.not. (feltétel)) goto L
          return
          goto L+1
  L       continue
          a = b
  L+1 continue
```

Egyes fordítóprogramok ellenőrzik, hogy minden utasítás elérhető-e. Jelen esetben úgy döntenek, hogy „**goto L+1**" nem az, és kiírnak egy figyelmeztetést. Ugyanilyen figyelmeztetést kapunk, ha a **return** helyére **break**-et, **next**-et, **goto**-t vagy **stop**-ot teszünk.

A javítás egyszerű: hagyjuk el az **else**-t. Az **else** a legszigorúbban véve is felesleges, és ez okozza az elérhetetlen utasítás generálását is. Ennek ellenére sajnáljuk elhagyni, mivel ez megfosztana minket egy fontos, jól bevált formától, és befolyásolhatná az áttekinthetőséget. Ha egy **else**-t látunk, tudjuk, hogy a két utasítás közül csak az egyik lesz végrehajtva. Ha azonban az **if**-et egy második utasítás is követ, ez azt sugallja, hogy a második utasítás is végrehajtódik. A szövegkörnyezet alapján meggyőződhetünk az ellenkezőjéről, a forma mindenképpen félrevezető.

A könyvben, ahol csak lehetett, mindenhol **else**-t és **else if**-et használtunk annak hangsúlyozására, hogy csak az egyik utasítássorozatot kell végrehajtani. Ugyanakkor elkerültük az **else** használatát olyan esetekben, amikor ebből hozzáférhetetlen utasítás jöhet létre, ezzel is javítva a programok portabilitását.

Gyakorlat

9.1 Miért nem jó, ha minden címke után egy **continue** utasítást generálunk? *Útmutatás:* melyik az a FORTRAN utasítás, amelyhez mindig tartozik címke, de nem kell végrehajtani? □

9.4 AZ ELEMZÉS

Most rátérünk az elemzésre, amely azt biztosítja, hogy a kódgenerálási műveletek a megfelelő időben és a megfelelő értékek alapján végződjenek el. Az eljárás lényege a következő: Amikor egy utasítás elejéhez (**if, else, while, do,** bal kapcsos zárójel, számjegyek) érünk, a megfelelő típust egy veremben tároljuk és behívjuk a típusnak megfelelő kódgeneráló rutint, ha ilyen létezik (pl. **if** esetén **ifcode**-ot). Az **if, do** és **while** kulcsszavakhoz tartozó rutinok egy-egy egyedi címkét adnak vissza, melyeket egy párhuzamosan vezetett másik címkeveremben tárolunk.

Az utasítás végének elérésekor (**other, break, next** típusok, illetve jobb kapcsos zárójel vagy pontosvessző), behívjuk a megfelelő kódgeneráló rutint, ha ilyen létezik. Mivel az utasítás végét már megtaláltuk, az elemző esetleg a veremből is kiüríthet néhány elemet. Például

```
if (a)
    if (b)
        i = i + 1
j = j + 1
```

esetén az **i=i+1** elérésekor (**other** típusú), a veremből mindkét **if** kiüríthető; mivel nincs **else,** mindkét **if** utasítás befejeződött.

Az elemző rutin egyértelműen a program legbonyolultabb része, vázlata azonban elég egyszerű:

```
while (token ¬= EOF)
    if (token == if, else, while, do, bal kapcsos, számjegyek) {
        hajtsd végre a megfelelő kódgeneráló rutint
        vermeld a kódgeneráló által visszaadott szimbólumtípust és
        címkét}
    else {
        végezd el a kódgenerálást a megfelelő típusra
        (az csak other, break, next, pontosvessző, jobb kapcsos lehet)
        while (a verem nem üres) {
            if (verem == bal kapcsos)
                break
            if (verem == IF & következő szimbólum == ELSE)
                break
            a vermelt típus végénél esedékes kódgenerálás végrehajtása
            a verem ürítése
            }
        }
```

Ha eltekintünk a hibás forrásszöveg elleni védekezéstől, a fő problémát az okozza, hogy egy szimbólummal előre tartva kell haladnunk, hogy lássuk, vajon van-e **else** egy **if** utasításban. Tekintsük a következő utasítást:

```
if ( a )
    if ( b )
        c
    else
        d
```

Melyik **if**-hez tartozik az **else**? Ez a RATFOR-ban több más nyelvhez hasonlóan nem egyértelmű, logikailag bármelyikhez tartozhatna. Mi a széles körben elterjedt és jobban használható értelmezést választottuk, nevezetesen azt, hogy az **else** az őt megelőző legközelebbi, **else**-zel még nem párosított **if**-hez tartozik. Ezért, ha a verem tetején pillanatnyilag egy **if** van és a következő bemenő szimbólum egy **else**, az elemző nem ürítheti tovább a vermet, hanem tárolnia kell benne az **else**-t, amelyet később majd az **if**-el együtt kell kiüríteni.

Elemzési példaként fordítsuk le a következőt:

```
if ( a ) {
    if ( b )
        c
    while ( d )
        e
    }
else
    f
g
```

Először is az

```
if (.not. (a)) goto L1
```

utasítás generálódik, a verem tetejére pedig egy **if** kerül, az **L1** címkével együtt. A bal kapcsos zárójel a verem második, az újabb **if** pedig a verem harmadik helyére kerül, és az

```
if (.not. (b)) goto L2
```

utasítás generálódik.

Ezt követőleg a **c** feldolgozása következik, majd ha a **c**-hez tartozó kód generálása befejeződött, a verem tetején egy **if** áll, a következő szimbólum pedig egy **while**, ezért az **if** üríthető a veremből és generálható az

```
L2 continue
```

utasítás. A verem mutatójának értéke most 2, a verem tetején pedig egy bal kapcsos zárójel áll. A **while**-t a 3. pozícióban helyezzük el a veremben, és kivisszük a **while** kezdetéhez tartozó

```
   continue
L3 if (.not. (d)) goto L3+1
```

utasításokat. Amikor az **e** utasítás megérkezik, létrejön a neki megfelelő kód. Ezután a **while** üríthető a veremből és generálható a hozzátartozó befejező kód:

```
      goto L3
L3+1  continue
```

Ezután a verem második helyén a bal kapcsos zárójel áll, inputként pedig egy jobb kapcsos zárójel következik. Ekkor kiürítjük a veremből a bal zárójelet, a veremmutató értéke 1, inputként pedig egy **else** következik, amit tehát a verem 2. pozíciójában tárolunk és generáljuk a

```
   goto L1+1
L1 continue
```

utasítást. Végül a **g** kódot generáljuk.

Néhány ehhez hasonló kézi szimulációval meggyőződhetünk arról, hogy eljárásunk csakugyan úgy viselkedik, ahogyan elvárjuk tőle, és egyben segít annak áttekintésében, hogy hogyan *kell* viselkednie.

Most már abban a helyzetben vagyunk, hogy bemutathatjuk az elemző programját. Bár az első pillantásra tekintélyes méretű rutinnak látszik, mégis pontosan követi a korábban adott vázlatot.

```
# parse — RATFOR forrásprogram elemzése
   subroutine parse
   character lexstr(MAXTOK)
   integer lex
   integer lab, labval(MAXSTACK), lextyp(MAXSTACK), sp, token

   call initkw       # kulcsszavak elhelyezése a táblázatban
   sp = 1
   lextyp(1) = EOF
   for (token = lex(lexstr); token ¬= EOF; token = lex(lexstr)) {
```

```
        if (token == LEXIF)
            call ifcode(lab)
        else if (token == LEXDO)
            call docode(lab)
        else if (token == LEXWHILE)
            call whilec(lab)
        else if (token == LEXDIGITS)
            call labelc(lexstr)
        else if (token == LEXELSE) {
            if (lextyp(sp) == LEXIF)
                call elseif(labval(sp))
            else
                call synerr("hibás else.")
            }
        if (token==LEXIF|token==LEXELSE|token==LEXWHILE
        |token==LEXDO|token==LEXDIGITS
        | token==LBRACE) {
            sp = sp + 1         # utasítás kezdete
            if (sp > MAXSTACK)
                call error("veremtúlcsordulás az elemzőben.")
            lextyp(sp) = token        # a típus és az érték vermelése
            labval(sp) = lab
            }
        else {      # utasítás vége—felkészülés a verem ürítésére
            if (token == RBRACE) {
                if (lextyp(sp) == LBRACE)
                    sp = sp - 1
                else
                    call synerr("hibás jobb kapcsos.")
                }
            else if (token == LEXOTHER)
                call otherc(lexstr)
            else if (token == LEXBREAK | token == LEXNEXT)
                call brknxt(sp, lextyp, labval, token)
            token = lex(lexstr)    # mi is következik?
            call pbstr(lexstr)
            call unstak(sp, lextyp, labval, token)
            }
        }
    if (sp ¬= 1)
        call synerr("váratlan EOF.")
    return
    end
```

Az **initkw** a későbbi azonosítás céljából elhelyezi a kulcsszavakat a táblázatban. Ehhez — a **define**-hoz hasonlóan — az **instal**-t használja.

Mint már mondtuk, ha a **lex** által visszaadott szimbólum egy utasítás kezdetét jelzi, behívásra kerül a megfelelő kódgeneráló rutin. A típus és az esetleges generált címkék a veremben tárolódnak. A kódgeneráló rutinokra rövidesen visszatérünk.

Ha a beolvasott szimbólum egy utasítás végét jelzi, végrehajtjuk a megfelelő kódgenerálást és ahány elemet csak lehet, kiürítünk a veremből. Ezt az **unstak** végzi el:

```
# unstak — a verem ürítése az utasítás végén
    subroutine unstak(sp, lextyp, labval, token)
    integer labval(MAXSTACK), lextyp(MAXSTACK), sp, token

    for ( ; sp > 1; sp = sp - 1) {
        if (lextyp(sp) == LBRACE)
            break
        if (lextyp(sp) == LEXIF & token == LEXELSE)
            break
        if (lextyp(sp) == LEXIF)
            call outcon(labval(sp))
        else if (lextyp(sp) == LEXELSE) {
            if (sp > 2)
                sp = sp - 1
            call outcon(labval(sp)+1)
            }
        else if (lextyp(sp) == LEXDO)
            call dostat(labval(sp))
        else if (lextyp(sp) == LEXWHILE)
            call whiles(labval(sp))
        }
    return
    end
```

A megfelelő kódgeneráló rutin behívása után az **unstak** egy elemet kiürít a veremből. Ha a verem tetején **else** áll, akkor két elemet, mégpedig az **else**-t és a hozzá tartozó **if**-et kell kiüríteni.

A RATFOR főprogramja meghívja a **parse**-t. Az adott környezettől függően az azután behívhatja a FORTRAN fordítót a fordítás befejezése céljából.

```
# ratfor — RATFOR főprogram
    call parse
    stop
    end
```

Egy nyelvi fordító tervezésekor az egyik legnagyobb probléma annak eldöntése, hogy külön-külön mit csinálunk azzal a rengeteg szintaktikus hibával, ami a forrásprogramban előfordulhat. Ritkán kielégítő, ha az első hiba jelzése után abbahagyjuk a fordítást. Még ha nincs is esély arra, hogy az eredmény használható legyen, célszerű egy menetben a lehető legtöbb hibát felderíteni. Ez megköveteli, hogy a fordító a hiba után minél hamarabb visszakerüljön a saját kerékvágásába, és minél kevesebb másodlagos (következmény-) hibát jelezzen.

Ez bonyolult feladat. Bár a **parse** és az **unstak** tartalmaz némi védelmet a hibás forrásszöveg ellen — mint pl. hiányzó **if**-ek és kapcsos zárójelek kijelzése — ezek korántsem tökéletesek. Többek között egy hiba az elemzőt olyan állapotba hozhatja, hogy a további input-ot hibásnak érzékeli akkor is, ha az nem az, és ilyenkor hiba-

üzenetek tömege jelenik meg. Sok elemző ugyanígy viselkedik, bár ez egyáltalán nem kívánatos. Az egyik gyakorlat a hibakezelés tökéletesítésével foglalkozik.

Tudomásul kell vennünk, hogy elemzőnk semmilyen értelemben sem a lehető legjobb. Igaz, hogy komoly méretű nyelvek elemzői is előállíthatók kézi úton, ez azonban nagy munka és a kapott elemző gyakran nehezen módosítható, komoly hibákat tartalmaz vagy a tervezettől eltérő módon viselkedik. Szerencsére a RATFOR nyelvtana igen egyszerű. Ha egy nyelvet formálisan definiáltunk egy nyelvtan segítségével (ahogyan a legtöbb nyelvet definiálni kellene), elemzőjének elkészítése lényegében mechanikus feladat. Mint már említettük, ez az elemző generátorok („compiler—compiler"-ek) feladata, amelyek méretük és speciális jellegük miatt itt nem tárgyalhatók. (Lásd a fejezet végén levő bibliográfiai megjegyzéseket.) A mi elemzőnk eredeti változatát egy elemző generátor segítségével fejlesztették ki. Az itt bemutatott elemzőt ebből származtattuk vissza, miután a nyelv tervezését kielégítőnek találtuk.

Gyakorlatok

9.2 Módosítsuk a **parse**-t úgy, hogy ésszerűbben reagáljon a végzetes szintaktikus hibákra! Olyasmire gondolhatunk, hogy átugorjuk az inputot addig, ameddig egy biztos helyre nem érünk, ahonnan a fordítás már folytatható. (Hol lehetne ez?) Próbálkozhatunk kimaradtnak ítélt szimbólumok *beszúrásával* is. Milyen veszéllyel jár ez a megoldás? □

9.3 Egy érdekes lehetőség a hibakezelést *szerkesztési* folyamatnak tekinti, amely az elfogadhatatlan forrásszöveget beszúrások, helyettesítések és törlések sorozatával hibátlanná alakítja. Egy lehetséges elmélet szerint az a legjobb hibakezelés, amely minimális számú szerkesztési lépést igényel. Elemezzük ennek a kritériumnak a következményeit! A **parse** által felderített hibákhoz határozzuk meg a legrövidebb szerkesztési lépéssorozatokat! □

9.5 KÓDGENERÁLÁS

A **parse** és az **unstak** jó néhány kódgeneráló rutint hív. Most ezekkel foglalkozunk.

if *utasítás:*

Amikor egy **if** utasítás jelenik meg a forrásszövegben, el kell különítenünk a „feltétel" részt (ami egy kiegyensúlyozott zárójeles szerkezetben álló karakterlánc), két szomszédos, egyedi **L** és **L+1** címkét kell generálnunk és ki kell vinnünk az

if (.not. *(feltétel)***) goto L**

utasítást, majd vissza kell adnunk **L**-et. Ezt az **ifcode** végzi el:

```
# ifcode — az if kódja első részének generálása
    subroutine ifcode(lab)
    integer labgen
    integer lab

    lab = labgen(2)
    call ifgo(lab)
    return
    end
```

A **labgen** hozza létre a címkéket; rövidesen erre is rátérünk. Az **ifgo** a következő szerkezetű utasítást generálja:

if (.not. *(feltétel)*) **goto lab**

Ez adja a legtöbb munkát az output rutinoknak, melyeket pillanatnyilag csak a nevükkel és egy rövid megjegyzéssel jellemzünk, de rövidesen visszatérünk rájuk.

```
# ifgo — az "if(.not.(...)) goto lab" generálása
    subroutine ifgo(lab)
    integer lab
    string ifnot "if(.not."

    call outtab               # elmegyünk a 7. oszlopig
    call outstr(ifnot)        # "if(.not."
    call balpar               # a feltétel leválasztása és kivitele
    call outch(RPAREN)        # ")"
    call outgo(lab)           # "goto lab"
    return
    end
```

A **balpar** összegyűjti és kiviszi az **if** feltétel részét, ami egy kiegyensúlyozott zárójeles szerkezetben álló karaktersorozat. Ha a feltétel több sort foglal el, a **balpar** a feldolgozás során figyelmen kívül hagyja az újsor-karaktereket.

```
# balpar — kiegyensúlyozott zárójeles sorozat másolása
    subroutine balpar
    character gettok
    character t, token(MAXTOK)
    integer nlpar

    if (gettok(token, MAXTOK) ¬= LPAREN) {
        call synerr("hiányzó bal zárójel.")
        return
        }
    call outstr(token)
    nlpar = 1
    repeat {
        t = gettok(token, MAXTOK)
        if(t==SEMICOL|t==LBRACE|t==RBRACE|t==EOF) {
```

```
                call pbstr(token)
                break
                }
            if (t == NEWLINE)       # újsor-karakterek törlése
                token(1) = EOS
            else if (t == LPAREN)
                nlpar = nlpar + 1
            else if (t == RPAREN)
                nlpar = nlpar - 1
            # különben semmi különleges
            call outstr(token)
            } until (nlpar <= 0)
        if (nlpar ¬= 0)
            call synerr("hiányzó zárójel a feltételben.")
        return
        end
```

Mint már említettük, a RATFOR nem túlságosan ismeri a FORTRAN-t. A **balpar** nem csinál mást, mint összegyűjt egy karakterláncot. Nem végez szintaktikai ellenőrzéseket, eltekintve némi elemi, de hatékony védekezéstől az egyik leggyakoribb hiba, a kiegyensúlyozatlan zárójeles szerkezet ellen.

Azok a tevékenységek, melyeket egy **else** kezdetén, illetve egy **if** vagy **else** végénél kell elvégezni, egyenként csupán néhány sorból állnak. Ha **else** érkezik és a verem tetején egy **if** áll, az elemző behívja az **elseif**-et, amely a következő eredményt adja:

```
     goto L+1
L    continue
```

ahol **L** a veremben levő címke. Az **elseif** a következőképpen fest:

```
   # elseif — kódgenerálás az if végén, else előtt
      subroutine elseif(lab)
      integer lab

      call outgo(lab+1)
      call outcon(lab)
      return
      end
```

Az **outcon** egy címkézett **continue** utasítást generál, az **outgo** pedig egy adott címkére hivatkozó **goto**-t.

Egy **else** vagy egy **else** nélküli **if** befejezésekor az output-ot az **unstak** következő utasításai állítják elő:

```
if (lextyp(sp) == LEXIF)
      call outcon(labval(sp))
else if (lextyp(sp) == LEXELSE) {
      if (sp > 2)
            sp = sp - 1
      call outcon(labval(sp)+1)
      }
```

A két utóbbi rutin egy-egy címkézett **continue**-t állít elő, és a megfelelő címkét a veremből veszi elő. A **sp** értékének vizsgálata a hibakezeléshez tartozik, megakadályozza a verem nemkívánatos, hibás kiürítését, ha egy **if** nélküli **else** fordul elő.

A **labgen** egy sorozat különböző, folyamatosan egymás után számozott címkét generál és ezek közül az elsőt adja vissza függvényértékként. A generált címkék önkényesen 23 000-nél kezdődnek, ami valószínűleg nem vezet a RATFOR program írója által használt címkékkel való ütközéshez.

```
# labgen — n egymás utáni címke generálása, a függvény értéke az első
      integer function labgen(n)
      integer label, n
      data label /23000/

      labgen = label
      label = label + n
      return
      end
```

Címkék:

Ha egy címke érkezik (lexikális típusa **LEXDIGITS**), azt ki kell vinni annyi szóközzel kiegészítve, hogy az utasítás további része a 7. pozícióra kerüljön.

```
# labelc — címkekivitel
      subroutine labelc(lexstr)
      character lexstr(ARB)
      integer length

      if (length(lexstr) == 5)      # figyelmeztetés a 23xxx alakú címkére
            if (lexstr(1) == DIG2 & lexstr(2) == DIG3)
                  call synerr("figyelem: esetleges címkeütközés.")
      call outstr(lexstr)
      call outtab
      return
      end
```

Mivel RATFOR programokban címkékre — a **format** utasításoktól eltekintve — nincsen szükség, 23 000-rel kezdődő címke valószínűleg soha nem fordul elő. Biztonság kedvéért azonban a **labelc** figyelmeztet a 23 000-es nagyságrendű beolvasott címkék jelenlétére. Ugyanis ezt a szinte abszurd esetet sem hagyhatjuk figyelmen kívül, azonban annak pontos ellenőrzése, hogy valamelyik beolvasott címke nem egyezik-e egy generált címkével, az előfeldolgozót jóval bonyolultabbá tenné, de

csak keveset javítana rajta. A figyelmeztetés az ésszerű középút. Figyeljük meg, hogy csak a **labgen** és a **labelc** ismeri a generált címkék értékintervallumát, mivel semmi nem indokolja, hogy ezzel az előfeldolgozó más részei is törődjenek.

do *utasítás:*

A **do**-hoz tartozó kód generálása kézenfekvő. Leválasztjuk a **do** határait, generálunk két szomszédos **L** és **L+1** címkét, kivisszük a

do L *határok*

utasítást, és **L** értékét visszaadva fejezzük be a generálást.
Mivel a RATFOR lényegében nem ismeri a FORTRAN-t, a „határok" akármi lehet, amit a helyi FORTRAN fordító elfogad; leggyakoribb az **i=1, n** jellegű szerkezet. Ha ez a komponens bármely okból is hibás lenne, a hibát majd a FORTRAN fordító jelzi.

A **do**-hoz tartozó utasítás végén kivisszük a következő parancsokat:

```
L     continue
L+1   continue
```

Az **L+1** címkéjű **continue** utasítás a **do**-ban előforduló esetleges **break** kijárati pontja. (Az **L** a **next** utasítás esetén tesz hasonló szolgálatot.) A **do** utasítás megjelenésekor a **docode** kerül behívásra:

```
# docode — a do-hoz tartozó kód első részének generálása
   subroutine docode(lab)
   integer labgen
   integer lab
   string dostr "do"

   call outtab
   call outstr(dostr)
   lab = labgen(2)
   call outnum(lab)
   call eatup
   call outdon
   return
   end
```

Az **eatup** begyűjti az utasítás maradékát, azt a részt, amely a **lex** által leválasztott szimbólum után áll. Közönséges utasítások esetén az **eatup** valósítja meg a folytatási konvenciót is — a vesszőre végződő sorok folytatódnak. A **balpar**-hoz hasonlóan figyeli a kiegyensúlyozatlan zárójeles szerkezetet, de csak egy soron belül. Egy szokásos utasításból hiányzó zárójel nagyobb valószínűséggel hiba, mint szándékos sorfolytatás.

```
# eatup — az utasítás maradékának feldolgozása; folytatási sorok
# kezelése
   subroutine eatup
   character gettok
   character ptoken(MAXTOK), t, token(MAXTOK)
   integer nlpar

   nlpar = 0
   repeat {
      t = gettok(token, MAXTOK)
      if (t == SEMICOL | t == NEWLINE)
         break
      if (t == RBRACE) {
         call pbstr(token)
         break
         }
      if (t == LBRACE | t == EOF) {
         call synerr("váratlan kapcsos zárójel vagy EOF.")
         call pbstr(token)
         break
         }
      if (t == COMMA) {
         if (gettok(ptoken, MAXTOK) ¬= NEWLINE)
            call pbstr(ptoken)
         }
      else if (t == LPAREN)
         nlpar = nlpar + 1
      else if (t == RPAREN)
         nlpar = nlpar - 1
      call outstr(token)
      } until (nlpar < 0)
   if (nlpar ¬= 0)
      call synerr("kiegyensúlyozatlan zárójelek.")
   return
   end
```

A **dostat**-ot a **do** befejezésekor kell hívni, a két megfelelően címkézett **continue** utasítás kivitele céljából:

```
# dostat — do végéhez tartozó kód generálása
   subroutine dostat(lab)
   integer lab

   call outcon(lab)
   call outcon(lab+1)
   return
   end
```

while *utasítás:*

Egy **while** utasítás felismerésekor leválasztjuk a feltételt az **ifgo**-val és generáljuk a **next**, illetve a **break** utasításokhoz szükséges címkéket. Emlékeztetünk rá, hogy a következő utasítások kerülnek kivitelre:

```
      continue
L     if (.not.(feltétel)) goto L+1
                utasítás
                goto L
L+1 continue
```

Ezt a **whilec** és a **whiles** hozza létre:

```
# whilec — while elejéhez tartozó kód generálása
    subroutine whilec(lab)
    integer labgen
    integer lab

    call outcon(0)
# címkézetlen continue, hátha volt címke
# a while előtt
    lab = labgen(2)
    call outnum(lab)
    call ifgo(lab+1)
    return
    end
```

A **whiles** a **while**-hoz tartozó utasítás végén kerül végrehajtásra:

```
# whiles — while végéhez tartozó kód generálása
    subroutine whiles(lab)
    integer lab

    call outgo(lab)
    call outcon(lab+1)
    return
    end
```

A **break** *és* **next** *utasítások*

A **break** és **next** számára történő kódgenerálás azon alapszik, hogy a **do** és a **while** utasítások feldolgozásának során megfelelő címkék már bekerültek a verembe. A tárolt érték mindig a **next** címkéje, a **break**-é pedig ennél nagyobb. A **brknxt** megkeresi a veremben azt a **do** vagy **while** utasítást, amely a **next**-et vagy **break**-et tartalmazza, és kiadja a

goto L

utasítást, ahol **L** a megfelelő címke.

```
# brknxt — break és next kódjának generálása
   subroutine brknxt(sp, lextyp, labval, token)
   integer i, labval(MAXSTACK), lextyp(MAXSTACK), sp, token

   for (i = sp; i > 0; i = i - 1)
        if (lextyp(i) == LEXWHILE | lextyp(i) == LEXDO) {
             if (token == LEXBREAK)
                  call outgo(labval(i)+1)
             else
                  call outgo(labval(i))
             return
             }
   if (token == LEXBREAK)
        call synerr("hibás break.")
   else
        call synerr("hibás next.")
   return
   end
```

Az **other** *típus:*

Az eddig felsorolt utasítástípusok egyikéhez sem tartozó utasításokat az **otherc** dolgozza fel, átmásolásukhoz az **eatup**-ot használva:

```
# otherc — közönséges FORTRAN utasítás kivitele
   subroutine otherc(lexstr)
   character lexstr(ARB)

   call outtab
   call outstr(lexstr)
   call eatup
   call outdon
   return
   end
```

Gyakorlatok

9.4 Az **eatup** biztosítja, hogy a vesszővel végződő sorok folytatódnak. Valósítsuk meg azt a konvenciót is, hogy az alsó kötőjelre végződő sorok is folytatódnak, de törlődik belőlük az alsó kötőjel. Ez lehetővé teszi, hogy bármely sort bármely ponton megszakítsunk, és új sorban folytassunk. Célszerű lenne-e ez az aritmetikai műveleti jelek, pl. +, − stb. után? Mi a helyzet a /-lel? □

9.5 Ha egy **do** ciklusban nincs **break** utasítás, a **dostat** által generált második **continue** felesleges. Hogyan lehetne ezeket kiküszöbölni? Megéri-e a munkát? Hogyan kezeli az Olvasó által használt FORTRAN fordító a felesleges **continue** utasításokat? Hát a címke nélkülieket? □

9.6 KIVITELI RUTINOK

A kivitelre kerülő sorokat karakterenként gyűjtjük az **outbuf** tömbben; az **outp** mutató az **outbuf**-ban utoljára elhelyezett karakter indexe. Ezeket a változókat a **coutln** nevű közös adatmezőben tartjuk:

```
common /coutln/ outp, outbuf(MAXLINE)
    integer outp          # outbuf utolsó foglalt pozíciója; kezdőért.=0
    character outbuf      # itt gyűjtjük a kimenő sorokat
```

Az **outdon** kiüríti **outbuf**-t, és nullára állítja **outp**-t. Behívására az egyes utasítások feldolgozásának végén kerül sor, valamint **outch**-ban az előző sor kivitele céljából, a folytatási sor kezdése előtt. Az **outdon** a RATFOR egyetlen rutinja, amely ténylegesen kivitelre kerülő szöveget állít elő.

```
# outdon — egy kimenő sor befejezése
    subroutine outdon
    include coutln

    outbuf(outp+1) = NEWLINE
    outbuf(outp+2) = EOS
    call putlin(outbuf, STDOUT)
    outp = 0
    return
    end
```

Az **outch** karaktereket helyez el **outbuf**-ban; hozzá tartozik a folytatási soroknak a helybeli konvenciók szerinti kezelése. A sorok hossza a standard FORTRAN-nak megfelelően 72 karakter. A 6. oszlopban álló, szóköztől és zérustól különböző karakter azt jelzi, hogy a sor az előző folytatása; erre mi csillagot használunk.

```
# outch — egy karakter átvitele a kimeneti pufferbe
    subroutine outch(c)
    character c
    integer i
    include coutln

    if (outp >= 72) {        # folytatási kártya
        call outdon
        for (i = 1; i < 6; i = i + 1)
            outbuf(i) = BLANK
        outbuf(6) = STAR
        outp = 6
        }
    outp = outp + 1
    outbuf(outp) = c
    return
    end
```

Az **outtab**-ot több rutin is használja arra a célra, hogy **outp**-t elléptesse a 6. oszlopig.

```
# outtab — átlépés a 6. oszlopon
    subroutine outtab
    include coutln

    while (outp < 6)
        call outch(BLANK)
    return
    end
```

Van még néhány további kiviteli rutin a közös szerkezetek előállítására, de ezek mindegyike az **outtab**-ot hívja egy új rutin elején, **outch**-t az utasítás fordításának felépítésére, az **outdon**-t a befejezéshez. Csupán ezen három rutinnak kell ismernie a FORTRAN utasítások formai sajátosságait, így az olyan jelentőségteljes számokat, mint a 6 és a 72. Még **outtab** is **outch**-t hívja, pedig közvetlenül is írhatna szóközöket az **outbuf**-ba.

Tekintsük a hátralevő kiviteli rutinokat: Az **outstr** kivisz egy láncot az **outch** ismételt hívásainak segítségével. Elvégzi az idézőjelek közé tevő karakterláncok standard FORTRAN-beli *n* **h** alakra való konvertálását is, amire esetleg az Olvasó FORTRAN fordítójánál nincs is szükség. Az **outstr** az a rutin, ahol bármi egyéb karakterkonverziót célszerű elvégezni, pl. a „>" cseréjét **.gt.**-re stb. Ezeket gyakorlatként az Olvasóra bízzuk.

```
# outstr — karakterlánc kivitele
    subroutine outstr(str)
    character c, str(ARB)
    integer i, j

    for (i = 1; str(i) ¬= EOS; i = i + 1) {
        c = str(i)
        if (c ¬= SQUOTE & c ¬= DQUOTE)
            call outch(c)
        else {
            i = i + 1
            for (j = i; str(j) ¬= c; j = j + 1)      # keresd a végét
                ;
            call outnum(j−i)
            call outch(LETH)
            for ( ; i < j; i = i + 1)
                call outch(str(i))
            }
        }
    return
    end
```

Az **outnum** a számokat az **itoc** segítségével belső ábrázolásból karakteres formára konvertálja és kiviszi őket az **outch**-val:

```
# outnum — tízes rendszerbeli szám kivitele
   subroutine outnum(n)
   character chars(MAXCHARS)
   integer itoc
   integer i, len, n

   len = itoc(n, chars, MAXCHARS)
   for (i = 1; i <=len; i = i + 1)
       call outch(chars(i))
   return
   end
```

Végül, az **outcon** és az **outgo** rendre az „**L continue**” és a „**goto L**” utasítást viszi ki, majd meghívja az **outdon**-t, a sornak az **outbuf**-ból való tényleges kivitele céljából.

```
# outcon — "n continue" kivitele
   subroutine outcon(n)
   integer n
   string contin "continue"

   if (n > 0)
       call outnum(n)
   call outtab
   call outstr(contin)
   call outdon
   return
   end

# outgo — "goto n" kivitele
   subroutine outgo(n)
   integer n
   string goto "goto"

   call outtab
   call outstr(goto)
   call outnum(n)
   call outdon
   return
   end
```

A RATFOR szubrutinfája nem különösebben bonyolult; az alábbiakban a fontosabb rutinokat ábrázoltuk. A kiviteli rutinok nincsenek rajta, mivel a szerkezeti bonyolultságot nem befolyásolják:

```
ratfor
    parse
        initkw
            instal
        lex
            gettok
                ngetc
                    getc
                putbak
            alldig
            lookup
        ifcode
            labgen
            ifgo
                balpar
                    gettok
        elseif
        docode
            labgen
            eatup
                gettok
        whilec
            labgen, ifgo
        labelc
        otherc
            eatup
        brknxt
        unstak
            dostat, whiles
```

9.7 BŐVÍTÉSEK

Ha az alapeszköz egyszer készen van, az természetesen sokféleképpen továbbfejleszthető. Ezek nem mindegyike éri meg a befektetett munkát, a gyakorlatokban mégis felsorolunk néhány lehetőséget. Legtöbbjük meglehetősen egyszerű, és az alapprogramnak csupán kézenfekvő kibővítését igényli.

Gyakorlatok

9.6 Valósítsuk meg a következő műveleti jelek fordítását:

```
>     .ge.
>=    .ge.
<     .lt.
<=    .le.
==    .eq.
¬=    .ne.
¬     .not.
&     .and.
|     .or.
```

□

9.7 Ha jelkészletünk olyan szegényes, hogy utasítások csoportosítására sem kapcsos, sem szögletes zárójelet nem használhatunk, módosítsuk a RATFOR-t úgy, hogy valamilyen más jelölést tegyen lehetővé. A **begin-end**, **do-end** és a **do-od** olyan lehetőségek, melyeket más nyelvekben alkalmaztak. Számításba jöhetnek rövidebb jelsorozatok is, mint pl. < <és> > **$** (és **$**). □

9.8 Építsük be az **include**-ot és a **define**-t vagy a **macro**-t ebbe a programba! Milyen okunk lehet ennek végrehajtására? Milyen érvek szólnak egy ilyen bővítés ellen? □

9.9 Egészítsük ki az utasításokat a **for** és a **repeat-until** utasítással. Egy **until** nélküli **repeat** végtelen ciklus lesz. Ne felejtsük el, hogy mivel az **until** elhagyható, a **repeat-until** ugyanúgy kétértelmű, mint az **if-else**. Kezeljük ezt hasonló módon! □

9.10 Programjaink közül néhányban célszerű lenne több ciklusból egyszerre kilépni (lásd pl. a 2. fejezetbeli **expand**-ot). Alakítsunk ki és valósítsunk meg egy többszintes **break** és **next** utasítást! Milyen előnyei és hátrányai lesznek ezeknek az utasításoknak az olvasható és könnyen módosítható programok írásának szemszögéből nézve? □

9.11 A 6. fejezetben említettük az **andif**, a 8.-ban pedig a neki megfelelő **orif** logikai műveletet. Ezek a logikai kifejezések balról jobbra történő kiértékelését, valamint azt biztosítják, hogy a kiértékelés befejeződik, amint a logikai érték ismertté válik. Valósítsuk meg az **andif**-et és az **orif**-et (**&&** és || célszerű rövidítések)! □

9.12 Bővítsük a nyelvet a **string** deklarációval! Az ANSI FORTRAN megköveteli, hogy a **data** utasítások csakis a deklarációk után forduljanak elő. Mennyire bonyolítja ez a problémát? □

9.13 Mind a RATFOR, mind a FORTRAN legnagyobb hiányossága, hogy csak nagyon korlátozott számú adattípus kezelésére képes. Hogyan lehetne karakter- és karakterlánc-változókat, struktúrákat és mutatókat bevezetni a nyelvbe? □

9.14 Sok azonosítónk erőltetett, mivel betartottuk a FORTRAN maximálisan 6 karakteres névhosszúsági előírását. Módosítsuk a RATFOR-t úgy, hogy a hosszabb neveket 6 karakteresre csonkítsa! Kell-e a RATFOR-nak ismernie a FORTRAN kulcsszavait? Finomítsuk a RATFOR-t azzal a biztonságosabb megoldással, hogy egyedi belső neveket generáljon azon csonkított azonosítók helyett, amelyeknek első hat karaktere azonos! □

9.15 Végezzünk méréseket néhány programon annak felderítésére, hogy egy **if** után milyen gyakran fordulnak elő vezérlésátadási utasítások (pl. **return, goto, break, next** és **stop**)! Mérjük meg futó programokon, hogy ezek milyen gyakran kerülnek végrehajtásra. Ha a mérések azt mutatják, hogy érdemes, írjuk át az elemzőt ezen speciális esetek optimalizálására! Érdemes-e az Olvasó szerint kiküszöbölni az el nem érhető kód generálásának lehetőségét? FORTRAN fordítónk milyen tulajdonságait kell kivizsgálnunk, mielőtt bármit is átírnánk? Miért nehezebb az

```
if (feltétel)
    i=0
```

alakú utasítás optimalizálása? Gondoltuk volna-e ezt előre? Érdemes-e megcsinálni? □

9.16 Javítsuk a RATFOR *hibakezelését!* Alakítsuk át az elemzőt úgy, hogy jobban kezelje az olyanféle hibákat, mint az **if** nélküli **else**, a hiányzó vagy feles-

leges kapcsos zárójel stb. Javítsuk a *hibafelderítő* képességet azzal, hogy gondosan ellenőrizzük a forrásprogramokat, nem tartalmaznak-e nyilvánvaló hibákat, mint pl. kiegyensúlyozatlan zárójeles szerkezet, szomszédos műveleti jelek stb. Javítsuk a hibák kijelzését azáltal, hogy minden kivitelre kerülő sort a neki megfelelő forrássor által meghatározott sorrendjelzővel látunk el. Ily módon a FORTRAN fordítóból érkező üzenetek könnyebben kapcsolhatók a RATFOR forrásszöveghez. (Gondoljunk rá, hogy ez alapvetően rendszerfüggő, bár a FORTRAN forrássorok 73...80 pozíciói lyukkártyás környezetben általában rendelkezésre állnak.) Ha az **include**-ot is beépítettük az előfeldolgozóba, hogyan jelezhetjük a beillesztett file-okban levő hibás sorok sorszámát? □

9.17 Javítsuk a generált program olvashatóságát megfelelő bekezdések alkalmazásával, a szóközök megtartásával, a felesleges **continue** utasítások kiküszöbölésével és a RATFOR kommentároknak FORTRAN kommentárokra való átalakításával! Megéri ez a fáradságot? □

9.18 Írjunk egy „csinosítási” programot, amely egy formázatlan RATFOR programot tetszetős külalakúvá formál bekezdések és megfelelő sorbeosztás segítségével. Mi célszerűbb: a csinosítási programot használni vagy sajátkezűleg kialakítani programjaink tetszetős külalakját? Elvárható-e, hogy már maga a RATFOR szép forráslistát produkáljon? Milyen kapcsolat van a kérdésre adott válasz és az Olvasó számítógépes környezete között? □

9.19 Az előfeldolgozónál sokkal bonyolultabb feladat egy „strukturáló” létrehozása. Ez olyan program, amely egy FORTRAN programot konvertál strukturált nyelvre — mint amilyen pl. a RATFOR. Tervezzünk meg egy ilyen programot több változatban, különböző szintű szolgáltatásokkal! Mennyire tökéletes munkát végezhet egy strukturáló? □

9.20 Más programozási nyelvek is javíthatók előfeldolgozással. (Ha azonnal nem jut eszünkbe ilyen példa, gondolhatunk az ALGOL, APL, assembly nyelv, BASIC, COBOL, PL/1, SNOBOL és a helybeli jobvezérlő nyelv különböző aspektusaira is.) Az Olvasó rendelkezésére álló operációs rendszer lehetővé teszi-e egyáltalán bármely nyelvben egy előfeldolgozó alkalmazását? Tervezzünk és valósítsunk meg egy előfeldolgozót, amely ésszerű mértéket tartva végez tökéletesítéseket! Tartsuk szem előtt, hogy sokkal hasznosabb, ha egy feladatnak egy részét oldjuk csak meg, de azt jól, mintha az egészet megoldanánk, de rosszul (vagy késve). □

9.21 Ha az Olvasó által használt rendszer megengedi, írjon olyan előfeldolgozót, amely beolvassa az általunk használt jobvezérlő nyelvet — file átirányítása a $<$ és $>$ operátorral, valamint a programcsatlakozás —, és létrehoz egy jobvezérlő programot tartalmazó file-t, amely azután lefuttatható. A programcsatlakozást közbülső file-ok segítségével valósítsuk meg. Mennyi támogatásra van szükség az operációs rendszertől? Mennyit kellene változtatni az előfeldolgozón annak érdekében, hogy beépíthessük egy interaktív rendszer parancsértelmezőjébe? □

9.8 MÉRÉSI EREDMÉNYEK

Ha egy, a RATFOR-hoz hasonló előfeldolgozót használunk, többe kerül a programok fordítása. De mennyivel? Ez függ a használt rendszertől, de durván becsülve, minden fordítás költsége duplázódik, mivel a programot kétszer olvassuk végig — egyszer RATFOR-ban és egyszer FORTRAN-ban, az egyetlen FORTRAN-ban való olvasás helyett.

Határozott érzésünk, hogy a többletköltség eltörpül az előfeldolgozó által nyújtott előnyök mellett. Még ha egy fordítás kétszer annyiba is kerül, tapasztalatunk szerint sokkal kevesebb fordításra van szükség, mivel a program *hamarabb válik működőképessé*. Ha egy hibát percek, és nem órák alatt találunk meg, ez sok fordítással felér. A másik megtakarítást az jelenti, hogy a már használatban levő programok módosítási költségei nagymértékben csökkennek. Valamennyi, e könyvben szereplő program több lényeges módosításon ment át. Szilárd meggyőződésünk, hogy ennyi módosítást nem lehetett volna standard FORTRAN-ban végrehajtani.

Mindezek ellenére érdemes utánanézni, hogy az előfeldolgozó hol tölti az időt, hogy megfontolhassuk a felgyorsítását. A jelen fejezetben szereplő RATFOR-t egy 900 soros program feldolgozása során mértük be. Az idő 65%-át a legalacsonyabb szintű — **getc** és **putlin** alatti — B/K-val töltötte. Az általunk használt rendszerben mind a **getc**, mind a **putlin** formázott FORTRAN B/K-t használ, ami hírhedten lassú.
A maradék 35% a RATFOR futási idejéből a következőképpen oszlott meg:

gettok	**11,1%**
ngetc	**5,2**
outstr	**3,2**
getc	**2,5**
type	**2,4**
outch	**2,0**
eatup	**1,5**
putbak	**1,4**
lookup	**1,1**

Minden más 1% alatt maradt. A tanulság újra az, hogy mindaddig, amíg nincs egy igazán gyors B/K rendszerünk, nyugodtan írhatunk olyan áttekinthető programot, amilyent csak tudunk, semmi nem számít. Ezután pedig a szimbólumkezelő **gettok** rutint célszerű megvizsgálni.

Bemértük a RATFOR egy gyakorlatban használatos változatát is, amely tartalmazza a **define**-t, és az **include**-ot és karakterkonverziót is végez. Ez idejének 60%-át töltötte B/K-val és 15%-át szimbólumoknak egy 60 szimbolikus konstansot tartalmazó definíciós táblában való keresésével. Ez alátámasztja azt a következtetésünket, hogy a B/K idő a domináns, és azt az állításunkat is, hogy a lineáris keresés gyakran elegendő.

Általánosságban megállapítható, hogy sok programnál a feldolgozás maga nem számottevő a karakterek puszta be- és kiviteléhez képest. A könyvben bemutatott eszközök legnagyobb része még hatékony B/K rutinok esetén is idejét jórészt B/K-ra fordítja. (A **find** és a **macro** néhány alkalmazása kivétel lehet.)

Az időmérések elvégzése gyakran nem könnyű, legalábbis operációs rendszerünkkel való gyakori összezörrenések nélkül nem az. Igen hasznos mérőeszköz, amely sokkal kevésbé szorul a rendszer segítségére, egy *számláló program,* amely megszámlálja, hogy a program egyes utasításai hányszor kerülnek végrehajtásra. A számlálást

a forrásprogramhoz a fordítás előtt illesztett utasítások végzik, amelyek a módosított program futása után az eredeti forráslista mellett az összegyűjtött információt is kinyomtatják. Még ha csupán az egyes utasítások végrehajtásának számát ismerjük is, már ebből is meghatározható, hogy a program mely része kerül leggyakrabban végrehajtásra, és nagy valószínűséggel ez emészti fel a legtöbb időt. Kimutatja, hogy a program mely részei nem hajtódnak végre, egyáltalán, ami felesleges programrészre, elégtelen kipróbálásra, vagy egyszerűen csak hibás programra utal. A végrehajtási időben jelentkező hibákat a szükségesnél többször végrehajtott részek alapján fedezhetjük fel. Ilyenek például a cikluson belüli számítások, amelyek akkor is végrehajtódnak, amikor nem lenne rájuk szükség.

Gyakorlat

9.22 Egy programozási nyelv előfeldolgozója majdnem minden eszközt a kezünkbe ad egy végrehajtásszámláló program kidolgozásához. Dolgozzuk ki a lehető legegyszerűbb végrehajtásszámlálót! Könnyebb-e végrehajtásszámlálót készíteni a RATFOR-hoz, mint a FORTRAN-hoz? □

9.9 MŰKÖDÉSI STATISZTIKA

Megvizsgáltuk azt is, hogy hogyan használják a RATFOR-t a gyakorlatban. Megszámláltuk az egyes utasítástípusok előfordulási gyakoriságát 5400 RATFOR programsorban (a könyv programjaiban), hogy lássuk, mennyire gyakran fordulnak elő, azaz érdemes volt-e őket beépíteni az előfeldolgozóba. Az eredmények a következők:

	összes	{ } áll utána	
if	**556**	**161**	
else	**229**	**81**	**(132 else if)**
for	**154**	**55**	
while	**45**	**15**	
repeat	**22**	**21**	
until	**14**		
do	**0**		
break	**55**		
next	**0**		

Természetesen a meglevő eszközeinket használtuk a statisztikák elkészítéséhez: a **find**-ot a kívánt sorok kikeresésére, majd a **translit**, a **charcount**, a **linecount** és az **edit** különböző kombinációit a sorok szétdarabolására és a számlálásra.

Összefoglalva, a vezérlőutasítások mindössze az összes sor 20%-át adják, a többi közönséges, régi FORTRAN. Mostanra azonban az Olvasó már egyet kell, hogy értsen azzal, hogy a RATFOR sokkal könnyebben olvasható, mint a FORTRAN; a különbség ennek a 20%-nak köszönhető. A ciklusok és az összetett utasítások száma durván azt mutatja, hogy hány **goto**-t és címkét küszöböltünk ki.

Legtöbb ciklusunk a feltételvizsgálatot a ciklus elején végzi — **for** és **while** kilencszer több van, mint **repeat**. (Elegánsan kezeljük a „ne csinálj semmit" esetét.) A **break** utasításra láthatólag szükség van, a **next** viszont sokkal kevésbé fontos; eredetileg ugyan elég sokszor alkalmaztuk, de ezek a program finomítása során természetes módon eltűntek.

Látjuk, hogy **do**-ra – legalábbis nekünk – nemigen volt szükségünk. Valójában, mivel nem találtunk egyetlen **do**-t sem, megvizsgáltuk a **for**, **while** és **repeat**

utasításokat a **do**-val való helyettesíthetőség szempontjából. A **for** utasítások mintegy negyed részét lehetne **do**-val pótolni, ha esetenként a határokat beállító utasításokat is beillesztenénk a programba, és *majdnem mindig* beépítenénk egy vizsgálatot, hogy a nullaszor futó ciklust megoldjuk. A **for** utasítások egy további negyed része erőszakkal valószínűleg **do**-vá alakítható, de csak az érthetőség feláldozásának árán. A többi **for**, valamint lényegében az összes **while** és **repeat** egyszerűen egyáltalán nem pótolható **do**-val. Ezek ciklusok, de ismétlődéseik nem aritmetikailag szabályozottak.

A tanulság világos: Ezen programgyűjteményben csupán néhány ciklust fejez ki legjobban a **do** utasítás (amely a FORTRAN egyetlen ciklusképző utasítása). Ha ragaszkodunk a **do** használatához, logikánkat a legtöbb esetben a **do**-hoz tartozó megszorításoknak kell alárendelni. Ilyen erőszakolt logika alkalmazása pedig nem teszi kellemessé a munkát.

BIBLIOGRÁFIAI MEGJEGYZÉSEK

A FORTRAN előfeldolgozók újabban igen népszerűvé váltak, valószínűleg azért, mert kis befektetéssel igen hasznosan alkalmazhatók. A legtöbbjük fejlettebb vezérlési szerkezeteket tartalmaz; néhány pedig a külalakkal kapcsolatos kérdésekkel foglalkozik. (A többség véleményétől eltérően úgy gondoljuk, hogy egy nyelv külső megjelenése is fontos.) Három, széleskörűen elterjedt előfeldolgozó, amely jóformán minden FORTRAN rendszerrel használható, a T. Beyer (Univ. of Oregon) által fejlesztett FLECS; a J. Cook és C. Zahn (Stanford Linear Accelerator) által kifejlesztett MORTRAN és az E. Miller (General Research) által fejlesztett IFTRAN. A MORTRAN egy makrofeldolgozón alapul és ezért könnyebben módosítható és bővíthető, mint egy fordítóprogramra alapozott megoldás. A RATFOR maga szintén hozzáférhető; a könyvben szereplő változat, amely az eredeti kiadótól beszerezhető géppel olvasható formában tartalmazza a **for**-t, a **repeat-until**-t, az **include**-ot és a **define**-t.

Igen terjedelmes irodalom áll a fordítási módszerek iránt komolyan érdeklődők rendelkezésére. Megismerhetjük A. V. Aho és S. C. Johnson „LR parsing" című cikkét (Computing Surveys, 1974. június) vagy A. V. Aho és J. D. Ullman „The Theory of Parsing, Translation and Compiling" c. művét (Prentice-Hall 1972.).

Előfeldolgozókat természetesen nemcsak hagyományos programozási nyelvekhez készítenek. Több előfeldolgozót készítettünk ahhoz a nyomdai szedő nyelvhez, amelylyel ezt a könyvet szedték; az egyik legsikerültebb egy matematikai szedőnyelvet realizál. (B. W. Kernighan és L. L. Cherry: „A system for typesetting mathematics", CACM 1975. március.)

Érdemes elolvasni D. E. Knuth: „An empirical study of FORTRAN programs" c. cikkét (Software-Practice and Experience, 1971 április), amely lebilincselően mutatja be, hogy mi mindent lehet egy utasításvégrehajtás-számláló programmal felderíteni, és leleplező statisztikákat tartalmaz a FORTRAN gyakorlati alkalmazásával kapcsolatban.

C. A. R. Hoare egyszer azt mondta: „[A nyelvtervezőnek] egy dolgot nem szabad tennie: bevenni a nyelvbe saját ki nem próbált ötleteit". Mi szigorúan tartottuk magunkat ehhez az intelemhez. A RATFOR vezérlőutasításait szégyenkezés nélkül elloptuk a *C* nyelvből, amelyet D. M. Ritchie fejlesztett a UNIX operációs rendszerhez. A RATFOR-ral (és a könyv többi programjával) kapcsolatos méréseket egy Honeywell 6070-en végeztük egy A. D. Hall által készített időmérő programcsomag felhasználásával.

Utószó

Hosszú utat tettünk meg. Kilenc, programokkal teletűzdelt fejezet sok megemésztendő anyagot jelent. Ha az Olvasó nem is tett mindent magáévá az első alkalommal, ne veszítse el a kedvét; erre nem is számítottunk. Még a legjobb programok teljes megértése is időt igényel, az összes következményeket pedig ritkán lehet átlátni anélkül, hogy megkísérelnénk azt használni vagy módosítani. A programozással kapcsolatos tudásunk legnagyobb része a program szövegével való közvetlen munkából ered: annak olvasásából, átdolgozásából és újraolvasásából.

Az *olvasás* és az *átdolgozás* a kulcsszavak. Egyetlen program sem tökéletes az első nyers változatában, bármilyen programozási módszerrel írtuk is azt meg. Bár a könyvben szereplő valamennyi rutint sokszor átírtuk, mégsem állítjuk egyikről sem, hogy tökéletes. A mélyenszántó átdolgozás költséges és időt rabló luxusnak tűnhet, de ha a program áttekinthető, és kisméretű modulokból áll, akkor nem az. Továbbá tapasztaljuk majd, hogy ha gyakorlatra teszünk szert a programok olvasásában és átdolgozásában, első programváltozataink egyre javulnak, mivel rövidesen rájövünk, hogy mit kell használni és mit kell elkerülni, mely módszerek jók és melyek rosszak. A programok átírása még ezután is a programozási folyamat fontos része marad.

A legtöbb esetben az átírás célja az, hogy a programot egyszerűsítsük, érthetőbbé tegyük és bonyolultságát elviselhető szinten tartsuk. A *bonyolultság kézbentartása* a programozás lényegéhez tartozik. A fejben tartható részletek száma mindig korlátozott lesz. Könyvünkkel jórészt azt igyekeztünk megtanítani, hogyan tarthatjuk kézben a bonyolultságot.

A legalsó szinten nagy gondot fordítottunk a vezérlési szerkezetek megválasztására és arra, hogy ezeket hogyan használjuk. Nem volt szükségünk például sem a **goto**, sem a **do** utasításra. Az **if** utasítást ritkán skatulyáztuk kettőnél nagyobb mélységben, eltekintve az **else if**-ek többkimenetelű döntések esetén való alkalmazásától. A ciklusok végére vonatkozó feltételvizsgálatot általában a ciklus elején végezzük, mielőtt késő lenne. A szubrutinok és a függvények ritkán haladják meg az egy oldalt és legtöbbjük ennél jóval rövidebb, Ennek köszönhetően a programok olvashatók. Könnyen meggyőződhetünk arról, hogy egy modul valószínűleg hibátlan, ha olyan részekre oszlik, amelyek egyszerre felfoghatók és sorban olvashatók.

Az egyes moduloknak nagy a *belső összetartó ereje:* jó oka van annak, hogy külön egységet alkossanak. Egyik modul sem összekuszált tevékenységek halmaza, sem pedig valamely más modul által képviselt funkció önkényesen kiszakított darabja. Ennek következtében minden egyes rutin feladata egy-két sorban leírható. Továbbá a rutinokat úgy írtuk, hogy a *specifikációjukat kielégítsék,* ami felette áll annak a megoldásnak, hogy először megírunk egy hasznosnak ígérkező szubrutint, majd leírjuk, hogy mit és hogyan hajt végre.

A könyvben több program áll 500...1000 forrássorból, de fogalmilag egy ilyen

program mégsem nevezhető „nagy”-nak. Modulonként vagy bekezdésenként haladva, mindegyik program megérthető. Ez azért van így, mert a szubrutinok hierarchiáját úgy terveztük, hogy egyik modul sem tud sokat a teljes feladatról és csupán néhány közvetlen szomszédjával áll kapcsolatban. Kicsiny a veszélye annak, hogy a hierarchia egy részében való változtatás váratlan következményekkel jár egy másik részben, mivel a modulok annyira függetlenek egymástól, amennyire csak lehet, a fennálló kapcsolatok pedig jól áttekinthetőek.

Igyekeztünk a programok módosítását megkönnyíteni azáltal, hogy a tervezési döntéseket és az adatszerkezeteket elfedtük úgy, hogy azok a rutinok, melyeknek nem kell ismerniük, nem is ismerik ezeket.

A programok ellenőrzéseket és olyan biztonsági intézkedéseket tartalmaznak, hogy a hibák és összeférhetetlenségek hamar kiderülnek. A jelkészletet, a paramétereket és a logikai jelző értékeit szimbolikus konstansokkal fejeztük ki, hogy értéküket egyetlen módosítással a teljes programban meg lehessen változtatni. Gondunk volt arra is, hogy az operációs rendszerrel való kapcsolatot a lehetőségekhez képest néhány elemi műveletre koncentráljuk, és így a programok nagy része a helyi környezettől független legyen.

Végül, a hierarchia felső szintjén olyan programokat írtunk, amelyek együttműködésre alkalmasak, így összetett feladatokat a meglevő programok *kombinálásával,* és nem új programok írásával lehet megoldani. Ilyenkor minden program egy modul, amelyet egyszerű módon tudunk más modulokkal összekapcsolni.

Ez a szó jó értelmében vett „strukturált programozás”. Látható, hogy a módszer eredményesen működik „éles” programok esetén is. Előnyei igen számottevőek: érthető, megbízható, erőteljes programokat tudunk írni, és emellett viszonylag kevéssé vagyunk érzékenyek a tervezési döntésekben jelentkező, esetenként lényeges változásokra vagy az egyik gépről a másikra való áttérésre. A minden szinten jelen levő megfelelő programszerkezet nem csak elvileg szép, hanem egy összetett feladat sikeres kézbentartásához nélkülözhetetlen is.

A programszerkezettel kapcsolatos megfontolások mellett megpróbálunk néhány hasznos tanácsot adni a programozási tevékenységhez is. Ezek, mint minden egyéni vélemény, vitathatók, de mi úgy találtuk, hogy beváltak.

Az első számú elv az, hogy a *program legyen minél egyszerűbb.* Minden szinten dolgozzunk olyan világosan, ahogy csak lehet és mindig a legegyszerűbb, legérthetőbb programot írjuk, amely megoldja a feladatot. Nem lehetünk természetesen végletesen naivak, a józan áttekintésre szükség van. Egy algoritmus kiválasztásakor kell, hogy bizonyos reményünk legyen arra, hogy az gazdaságos. Ha azonban a megvalósítás részleteit és stratégiáját elrejtjük, a meg nem felelő algoritmust anélkül cserélhetjük ki, hogy túl sok egyéb változtatást kellene eszközölnünk. Mivel eszközöket készítünk, nem szabad elfelejtkeznünk programjaink felhasználóiról, és az ő munkájukat kell saját munkánk nehezítése árán is, könnyebbé tennünk. Szerencsére az egységes, szabványos tervezés általában a jó felhasználói kapcsolatban is kifejeződik.

A második elv kapcsolódik az elsőhöz: a *fejlesztést több lépésben végezzük.* Az összetett feladatokat csak jól ellenőrizhető lépések formájában kezdjük el megoldani. Először a központi, a legfontosabb szempontokra koncentráljunk; ne foglalkozzunk a részletekkel. Ha a terv lényege jó, a későbbi bővítések jól fognak illeszkedni hozzá. Időközben az elkészült termék már használható, és a felhasználók tanácsai és tapasztalatai útmutatást jelentenek a további fejlesztés számára. Az is kiderülhet, hogy az elkészült rész önmagában véve is kielégítő. Ha egy feladat 90%-a már meg van oldva és jó, ez többet ér egy 99%-os megoldásnál, amelyet a jövő hónapra ígérnek.

A harmadik elv kissé meghökkentő: *hagyjuk, hogy a feladat nehezét más oldja meg.* Építsünk az általunk vagy mások által már elvégzett munkára, ahelyett, hogy minden alkalommal mindent elölről kezdenénk. Ha írunk egy rutint valamilyen céllal, tegyük ezt elég általánossá ahhoz, hogy hasonló feladatoknál is használható legyen. Nagyobb méretekben gondolkozva, gyakran nagy teljesítőképességű rendszerekhez juthatunk úgy, hogy egy kis programmal csatlakozunk egy nagyhoz — erre jó példa maga a RATFOR előfeldolgozó. És természetesen, amikor csak tehetjük, a munkát bízzuk a számítógépre, hiszen ez az eszközkészítés végső célja.

Egyetlen bonyodalom van, amelyet az Olvasó valószínűleg nem tud befolyásolni, és ez a helyi számítógépes környezet. De ha ez nagyon rossz is – ami sok esetben így van —, ebbe sem kell belenyugodnunk. A gyakran használt részek — mint pl. a programozási és jobvezérlő nyelvek — szerény módosítása is kifizetődik, ezért nem bocsátható meg, ha meg sem próbáljuk eltüntetni a rendszer legrosszabb tulajdonságait.

Mikor a programokat újra átnézzük, tartsuk szem előtt ezeket a gondolatokat. Bár javaslatainkat az egyes konkrét programok fejlesztése során adtuk, a belőlük leszűrhető tanulság általánosan is érvényes. Az itt összefoglalt tervezési elvek és programozási útmutatások olyan eszközök hatékony előállítására alkalmasak, amelyek megfelelően működnek, emberekkel és más programokkal való kapcsolatuk jó. Ez legyen az Olvasó célja bármily program írásánál is.

Függelék

Elemi műveletek és szimbolikus konstansok

Az elemi műveletek azok a rutinok, melyek a könyvben szereplő programokat összekapcsolják a futtatásukhoz használt operációs rendszerrel. A legtöbb program csak a **getlin**-t és **putlin**-t vagy a **getc**-t és **putc**-t igényli, melyek közül természetesen bármelyik pár megvalósítható a másik segítségével. A **getch** és a **putch** is ebbe a csoportba tartozik. Hasonlóan, az **error** és a **cant** is megírható a **remark** felhasználásával. A **remark**-ot csak azért nevezzük elemi műveletnek, mivel a FORTRAN-ban egy szubrutinnak átadott idézet végének felismerése a fordítótól függő módon történik.

Sok program használja a **getarg**-ot, melynek egy elfogadható ideiglenes változataként a paramétereket egy file-ból olvashatjuk be, minthogy minden program csak sorrendben és csak egyszer olvas paramétereket.

A file-rendszer és az operációs rendszer kapcsolatait az **open, close, create** és **remove** kezelik. Ezeket a 3. fejezet file programjai, a 4. fejezetbeli **sort** és a 6. fejezetbeli **edit** igényli. Az **edit** használja a **seek**-et és a **readf**-et is a munkafile-t alkalmazó változatban.

Egyes programok teljessé tételéhez a következő — a könyvben nem szereplő — rutinokra van szükség. Néhányuk FORTRAN-ban, rendszerint valamilyen más néven, rendelkezésre áll. Ezek: **abs (iabs), max (max0), min (min0)** és **mod.** Az **and, or** és **not** logikai függvényeket csak a **xor** és a **crypt** használja. Az **init** (a **format** esetén) és az **initkw** (a **ratfor** esetén) a paraméterek kezdőértékét a szövegben leírt módon állítja be.

A következő lista a könyvben használt szimbolikus konstansok értékének összefoglaló táblázata. Az *n* meg nem határozott numerikus értéket jelöl, általában egy veremként használt vagy egy karaktersorozatot tartalmazó tömb méretét. A lista nem tartalmazza a **MAX**-szal kezdődő neveket, ezek szintén numerikus értékek. Ugyancsak kihagytuk a **LET***x* és **DIG***n* alakú konstansokat; ezek rendre a megfelelő betű, illetve számjegy belső ábrázolását jelölik. Feltettük, hogy a karakterek belső kódolása kis pozitív egész számok alkalmazásával történt.

Paraméterértékekként, ahol csak lehetett, kinyomtatható karaktereket használtunk, hogy a programok és a belövés során kapott hibaüzenetek olvashatóbbak legyenek. A nem nyomtatható karaktereket dőlt, az aktuális számértékeket pedig álló betűvel szedtük.

ALPHA	a
AND	&
ANY	?
APPENDCOM	a
ARB	*n*
ARGFLAG	$
ARGSIZE	*n*
BACKSCAN	\
BACKSPACE	*visszalépés*
BACKWARD	−1
BLANK	*szóköz*
BOL	%
BOTTOM	62
BP	5
BR	3
BUFENT	5
BUFSIZE	*n*
CALLSIZE	*n*
CCL	[
CCLEND	]
CE	10
CHANGE	c
CHAR	a
CLOSIZE	4
CLOSURE	*
COLON	:
COMMA	,
COMMAND	.
COUNT	1
CURLINE	.
DASH	—
DEFTYPE	−4
DEL	d
DELCOM	d
DIGIT	0
DITTO	−3
DOLLAR	$
DQUOTE	”
ENTER	e
EOF	−1
EOL	$
EOS	−2
ERR	−3
ERROUT	7
ESCAPE	@
EVALSIZE	*n*
EXCLUDE	x
EXTR	x
FI	1
FO	13
FOLD	$
FORWARD	1
GLOBAL	g
HE	12
HUGE	1000
IFTYPE	−5
IN	7
INCTYPE	−6
INFILE1	1
INFILE2	2
INSERT	i
INSIZE	*n*
LASTLINE	$
LBRACE	{
LBRACK	[
LENG	4
LETTER	a
LEXBREAK	−8
LEXDIGITS	−9
LEXDO	−10
LEXELSE	−11
LEXIF	−12
LEXNEXT	−13
LEXOTHER	−14
LEXWHILE	−15
LINE0	0
LOGPTR	*n*
LPAREN	(
LS	4
MARGIN1	2
MARGIN2	2
MARK	2
MERGEORDER	7
MERGETEXT	*n*
MIDDLE	40
MINUS	—
MOVECOM	m
NAMESIZE	*n*
NCCL	n
NEXT	1
NEWLINE	*új sor*
NF	2
NFILES	*n*
NO	0
NOSKIP	+
NOSTATUS	0
NOT	¬
OK	−2
PAGELEN	66
PAGENUM	#
PAGEWIDTH	60
PERIOD	.
PL	14
PLUS	+
PREV	0
PREVCL	2
PRINT	P
PRINTCUR	=
PRINTFIL	f
QUIT	q
RBRACE	}
RBRACK	]
RCODE	0
READ	0
READCOM	r
READWRITE	2
RM	8
RPAREN	)
SCAN	/
SEEKADR	3
SEMICOL	;
SHARP	#
SKIP	*szóköz*
SP	6
SQUOTE	’
STAR	*
START	3
STDIN	5
STDOUT	6
SUBSTITUTE	s
SUBTYPE	7
TAB	*tabulátor*
TBL	t
TEXT	3
THRESH	5
TI	9
UL	11
UNDERLINE	=
UNKNOWN	0
UPD	u
WRITE	1
WRITECOM	w
YES	1

Rutinnevek mutatója

Ezen mutató a könyvben előforduló rutinok első soraként előforduló megjegyzéseket tartalmazza abc-sorrendbe rendezve. Az oldalszámozás a rutin végleges változatára vonatkozik.

ACOPY	- SIZE SZAMU KARAKTER MASOLASA FDI-BOL FDO-BA 110
ADDFIL	- A "NAME" FILE FELVETELE AZ ARCHIVUMBA 103
ADDSET	- C BEIRASA SET(J)-BE, HA BELEFER 69
ALLDIG	- AZ EREDMENY YES, HA STR CSUPA SZAMJEGYBOL ALL 301
AMATCH	- (NEM REKURZIV)-ILLESZKEDES KERESESE LIN(FROM)-TOL 156
AMOVE	- ATVITEL NAME1-BOL NAME2-BE 103
APPEND	- SOROK CSATLAKOZTATASA "LINE" UTAN 195
ARCHIVE	- FILE KARBANTARTO 99
BALPAR	- KIEGYENSULYOZOTT ZAROJELES SOROZAT MASOLASA 311
BRK	- AZ AKTUALIS FELTOLTOTT SOR VEGE 249

BRKNXT - BREAK ES NEXT KODJANAK GENERALASA 317

BUBBLE - V(1)'...V(N) BUBOREKOS RENDEZESE NOVEKVO SORRENDBE 117

CATSUB - HELYETTESITO SZOVEG CSATOLASA "NEW" VEGEHEZ 170

CENTER - EGY SOR KOZEPRE ALLITASA TIVAL SEGITSEGEVEL 253

CHANGE - CSERE "FROM"-ROL "TO"-RA 168

CHARCOUNT - KARAKTERSZAMLALAS A STANDARD INPUTON 24

CKGLOB - AZ ERINTETT SOROK MEGJELOLESE GLOBALIS ELOTAG ESETEN 212

CKP - PARANCS UTANI "P" VIZSGALATA 201

CLRBUF - (OPERATIV TARBELI) KEZDOERTEK UJ FILE ESETEN 198

CLRBUF - (MUNKAFILE) MUNKAFILE MEGSZUNTETESE 223

COMAND - FORMAZOPARANCS VEGREHAJTASA 235

COMPAR - LINBUF(LP1) ES LINBUF(LP2) OSSZEHASONLITASA 123

COMPARE - (EGYSZERU VALTOZAT) - FILE1 ES FILE2 OSSZEHASONLITASA 81

COMPARE - KET FILE OSSZEHASONLITASA 84

COMPRESS - SZOVEGTOMORITES A STANDARD INPUTROL 56

COMTYP - PARANCS DEKODOLASA 236

CONCAT - FILE-OK EGYESITESE A STANDARD OUTPUTON 89

COPY - KARAKTERMASOLAS 20

CRYPT - REJTJELEZES - REJTJELFEJTES 61

CTOI - KARAKTERSOROZAT ATALAKITASA EGESZ SZAMMA 75

DEFALT - SORSZAMOK ALAPFELTETELEZESENEK BEALLITASA 192

DEFINE - EGYSZERU JELSOROZAT-HELYETTESITESSEL MUKODO MAKROFELDOLGOZO 268

DELETE - FILE-OK TORLESE AZ ARCHIVUMBOL 110

DELETE - SOROK TORLESE "FROM"-TOL "TO"-IG 202

DETAB - TAB-OK KONVERTALASA MEGFELELO SZAMU SZOKOZRE 32

DIFMSG - ELTERO SOROK KIIRASA SORSZAMMAL EGYUTT 82

DOCMD - PARANCSOK FELDOLGOZASA, GLOBALIST KIVEVE (NEM TELJES) 194

DOCMD - PARANCSFELDOLGOZAS A GLOBALISOK KIVETELEVEL 214

DOCODE - A DO-HOZ TARTOZO KOD ELSO RESZENEK GENERALASA 314

DODASH - KOTOJELES PARAMETER KIFEJTESE 72

DODEF - DEFINICIOK BEVITELE A TABLAZATBA 284

DOGLOB - A LIN(I)-NEL LEVO PARANCS VEGREHAJTASA A JELOLT SOROKON 213

DOIF - VALASZTAS KET PARAMETER KOZUL 288

DOINCR - A PARAMETER NOVELESE EGGYEL 288

DOPRNT - "FROM"-TOL "TO"-IG TERJEDO SOROK NYOMTATASA 193

DOREAD - "FILE" OLVASASA "LINE"-TOL KEZDVE 209

DOSTAT - DO VEGEHEZ TARTOZO KOD GENERALASA 315

DOSUB - RESZSOROZAT KIVALASZTASA 289

DOWRIT - "FROM"-TOL "TO"-IG TERJEDO SZOVEG IRASA FILE-BA 210

EATUP - AZ UTASITAS MARADEKANAK FELDOLGOZASA; FOLYTATASI SOROK KEZELESE 315

EDIT - FOPROGRAM 217

ELSEIF - KODGENERALAS AZ IF VEGEN, ELSE ELOTT 312

ENTAB - SZOKOZOK HELYETTESITESE TAB-OKKAL ES SZOKOZOKKEL 48

EQUAL - STR1 ES STR2 OSSZEHASONLITASA 80

ESC - ARRAY(I) CSEREJE SPECIALIS KARAKTERRE, HA SZUKSEGES 71

EVAL - I.-J.-IG TERJEDO PARAMETEREK KIFEJTESE, A BEEPITETT FUGGVENYEK KIERTEKELESE, ILL. A DEFINICIOK VISSZAKULDESE 283

EXCHAN - LINBUF(LP1) ES LINBUF(LP2) FELCSERELESE 122

EXPAND - SZOVEGKIFEJTES A STANDARD INPUTROL 60

EXTRAC - FILE-OK KIEMELESE AZ ARCHIVUMBOL 109

FCOPY - FILEMASOLAS IN-BOL OUT-BA 89

FILARG - "NAME" KERESESE AZ ARGUMENTUM-LISTABAN 108

FILSET - ARRAY(I)-NEL KEZDODO FORDITASI HALMAZ KIFEJTESE DELIM-IG 69

FIND - MINTA KERESESE SZOVEGBEN 151

FORMAT - SZOVEGFORMAZO FOPROGRAM 235

FORMAT - SZOVEGFORMAZO FOPROGRAM (VEGLEGES VALTOZAT) 244

FPRINT - A "NAME" NEVU FILE LISTAZASA FIN-BOL 91

FSIZE - FILE MERETE KARAKTEREKBEN 105

FSKIP - N KARAKTER ATLEPESE AZ FD FILE-BAN 107

GETC - (EGYSZERU VALTOZAT) - KARAKTEROLVASAS A STANDARD INPUTROL 43

GETCCL - KARAKTER OSZT. KIFEJTESE ARG(I)-TOL PAT(J)-BE 163

GETDEF - (PARAMETER NELKULI ESET) NEVES DEFINICIO FELDOLGOZASA 269

GETFN - A FILE-NEV ATVETELE LIN(I)-TOL 209

GETFNS - FILE-NEVEK ATVETELE FNAME-BA, ISMETLODESVIZSGALAT 100

GETHDR - A FEJRESZ TARTALMANAK ATVETELE FD-BOL 107

GETIND - AZ INDEX MEGKERESESE A PUFFERBEN 198

GETLST - SORSZAMOK (HA VAN) ATVETELE LIN(I)-TOL, I NOVELESE 185

GETNUM - EGY TAG SORSZAMMA ALAKITASA 187

GETONE - EGY SORSZAMKIFEJEZES KIERTEKELESE 186

GETPAT - A PARAMETER ATALAKITASA MINTAVA 161

GETRHS - HELYETTESITO SOROZAT ATVETELE AZ "S" PARANCSHOZ 205

GETSUB - HELYETTESITO MINTA ATVETELE SUB-BA 169

GETTL - FEJ VAGY LAB MASOLASA BUF-BOL TTL-BE 242

GETTOK - ALFANUM. SOROZAT VAGY EGYETLEN NEM ALFAJEL OLVASASA A DEFINE SZAMARA 266

GETTOK - EGY SZIMBOLUM ELOALLITASA A RATFOR SZAMARA 298

GETTXT - (OPERATIV TARBELI) "LINE"-HEZ TARTOZO SZOVEG ELERESE ES ATVETELE 198

GETTXT - (MUNKAFILE) - "LINE"-HEZ TARTOZO SZOVEG ATVITELE TXT-BE 222

GETVAL - ESETLEGES NUMERIKUS PARAMETER KIERTEKELESE 237

GETWRD - EGY NEM URES SZO ATVETELE IN(I)-BOL OUT-BA I NOVELESEVEL 87

GNAME - N AZONOSITOJU EGYEDI FILE-NEV ELOALLITASA 130

GOPEN - LOW-TOL LIM-IG TERJEDO FILE-CSOPORT MEGNYITASA 130

GREMOV - LOW-TOL LIM-IG TERJEDO FILE-CSOPORT MEGSZUNTETESE 131

GTEXT - SZOVEGSOROK BEVITELE LINBUF-BA 120

HEAD - FEJRESZNYOMTATAS 92

HELP - HIBAUZENETEK KIIRASA 100

IFCODE - AZ IF KODJA ELSO RESZENEK GENERALASA 311

IFGO - AZ "IF(.NOT.(...))GOTO LAB" GENERALASA 311

INCLUDE - "INCLUDE" FILE HELYETTESITESE A FILE TARTALMAVAL 88

INDEX - C KERESESE AZ STR SOROZATBAN 64

INJECT - (OPERATIV TARBELI) SZOVEGBEILLESZTES CURLN UTAN 199

INJECT - (MUNKAFILE)-LIN BEILLESZTESE CURLN UTAN, IRAS A MUNKAFILE-BA 220

INSTAL - NEV ES DEFINICIO ELHELYEZESE A TABLAZATBAN 273

ITOC - EGESZ SZAM ATALAKITASA KARAKTERSOROZATTA 74

KWIC - A KULCSSZO ELOALLITASA 141

LABELC - CIMKEKIVITEL 313

LABGEN - N EGYMAS UTANI CIMKE GENERALASA, A FUGGVENY ERTEKE AZ ELSO 313

LEADBL - VEZETO SZOKOZOK TORLESE, TIVAL ALLITASA 246

LENGTH - KARAKTERSOROZAT HOSSZA 66

LEX - SZIMBOLUM LEXIKALIS TIPUSANAK MEGHATAROZASA 300

LINECOUNT - SORSZAMLALAS A STANDARD INPUTON 25

LOCATE - C KERESESE A KARAKTEROSZTALYBAN PAT(OFFSET)-NEL 159

LOOKUP - A NEV MEGKERESESE ES A DEFINICIO KIEMELESE A TABLAZATBOL 272

MACRO - PARAMETERES MAKROK KIFEJTESE 280

MAKECOPY - FILE ATMASOLASA EGY MASIK FILE-BA 96

MAKFIL - AZ N-HEZ TARTOZO UJ FILE LETREHOZASA 130

MAKHDR - FEJRESZKESZITES AZ ARCHIVUM TAGJAIHOZ 104

MAKLIN - (MUNKAFILE)- UJ SORHELY KIALAKITASA, A SZOVEG ATMASOLASA A MUNKAFILE-BA 221

MAKPAT - MINTAKESZITES ARG(FROM)-BOL, VEGJEL:DELIM 161

MAKSET - HALMAZ ELOALLITASA ARRAY(K)-BOL SET-BEN 69

MAKSUB - HELYETTESITO SOROZAT TAROLASA SUB-BAN 170

MATCH - ILLESZKEDES KERESESE BARHOL A SORON BELUL 152

MERGE - INFIL(1)...INFIL(NFILES) OSSZEFUTTATASA OUTFIL-BEN 132

MOVE - LINE1-TOL LINE2-IG TERJEDO SZOVEG ATVITELE LINE3 UTANRA 203

NEXTLN - A "LINE" UTANI SOR ELERESE 190

NGETC - EGY (ESETLEG MAR VISSZAKULDOTT) KARAKTER OLVASASA 267

NOTFND - "NINCS MEG" UZENET KIIRASA 108

OMATCH - EGYETLEN MINTA ILLESZTESI KISERLETE PAT(J)-NEL 157

OPTPAT - A LIN(I)-NEL ESETLEG TALALHATO MINTA FELEPITESE 188

OTHERC - KOZONSEGES FORTRAN UTASITAS KIVITELE 317

OUTCH - EGY KARAKTER ATVITELE A KIMENETI PUFFERBE 318

OUTCON - "N CONTINUE" KIVITELE 320

OUTDON - EGY KIMENO SOR BEFEJEZESE 318

OUTGO - "GOTO N" KIVITELE 320

OUTNUM - TIZES RENDSZERBELI SZAM KIVITELE 320

OUTSTR - KARAKTERLANC KIVITELE 319

OUTTAB - ATLEPES A 6. OSZLOPON 319

OVERSTRIKE - VISSZALEPESEKET TARTALMAZO SOROK HELYETTESITESE TOBBSZOROS SOROKKAL 52

PARSE - RATFOR FORRASPROGRAM ELEMZESE 307

PATSIZ - PAT N-NEL KEZDODO MINTAELEM HOSSZA 157

PBNUM - SZAM KONVERTALASA JELSOROZATTA ES VISSZAKULDESE AZ INPUTRA 288

PBSTR - EGY JELSOROZAT VISSZAKULDESE AZ AZ INPUT-RA 268

PFOOT - LAB KIVITELE 241

PHEAD - FEJ KIVITELE 240

PREVLN - A "LINE" ELOTTI SOR ELERESE 190

PRINT - FILE-OK LISTAZASA FEJLECCEL 90

PRINT - (ALAPFELTETELEK SZERINT BEMENET STDIN) - FILE-OK NYOMTATASA FEJLECCEL 94

PTEXT - SZOVEGSOROK KIVITELE LINBUF-ROL 121

PTSCAN - A MINTA KOVETKEZO ELOFORDULASANAK KERESESE 189

PUSH - EP ELHELYEZESE ARGSTK-BAN, UJ AP MUTATO VISSZAADASA 283

PUT - SOR KIVITELE MEGFELELO SORTAVOLSAGGAL ES BEKEZDESI MELYSEGGEL 240

PUTBAK - EGY KARAKTER VISSZAKULDESE AZ INPUT-RA 267

PUTBUF - BUF(1)...BUF(NSAVE) KIVITELE, NSAVE TORLESE 56

PUTC - (EGYSZERU VALTOZAT)- KARAKTERKIVITEL A STANDARD OUTPUTRA 43

PUTCHR - EGYETLEN KARAKTER KIVITELE VAGY VERMELESE 282

PUTDEC - DECIMALIS EGESZ KIIRASA LEGALABB W POZICION 73

PUTROT - KULCSSZOVAL KEZDODO SOROK LETREHOZASA 141

PUTTL - FEJ VAGY LAB SZOVEGSORANAK KIVITELE EGY ESETLEGES LAPSZAMMAL 241

PUTTOK - EGY SZIMBOLUM KIVITELE VAGY VERMELESE 282

PUTWRD - EGY SZO ELHELYEZESE OUTBUF-BAN 247

PUTWRD - EGY SZO BEHELYEZESE OUTBUF-BA, SORVEG-KIEGYENLITESSEL 250

QUICK - SZOVEGSOROK GYORSRENDEZESE 126

RATFOR - RATFOR FOPROGRAM 309

REHEAP - LINBUF(LINPTR(1)) BEILLESZTESE A MEGFELELO HELYRE 133

RELINK - KET KAPCSOLAS MODOSITASA 199
REPLAC - FILE-OK CSEREJE VAGY TORLESE 111
ROTATE - A PERMUTALT SOR KIVITELE 142
SCOPY - KARAKTERSOROZAT-MASOLAS
FROM(I) - TO(J) 104
SET - PARAMETERBEALLITAS ES
ERTEKELLENORZES 238
SETBUF - (OPERATIV TARBELI) A SORTAROLO PUFFER
KEZDO ALLAPOTANAK BEALLITASA 199
SETBUF - (MUNKAFILE) - MUNKAFILE LETREHOZASA,
0. SOR BEALLITASA 223
SETTAB - TAB-OK KEZDETI BEALLITASA 33
SHELL - V(1)...V(N) SHELL-RENDEZESE NOVEKVO
SORRENDBE 118
SHELL - SHELL-RENDEZES SZOVEGSOROKRA 121
SKIP - N URES SOR KIVITELE 92
SKIPBL - SZOKOZOK ES TAB-OK ATLEPESE
LIN(I)-TOL 187
SORT - SZOVEGSOROK RENDEZESE A TARBAN 120
SORT - SZOVEGSOROK KULSO RENDEZESE 129
SPACE - N URES SOR KIVITELE VAGY MEGALLAS A
LAP ALJAN 243
SPREAD - SZAVAK SZETTOLASA,
SORVEG KIEGYENLITES 250
STCLOS - LEZARTRA VONO BEJEGYZES ELHELYEZESE
PAT(J)-NEL 163

SUBST - "SUB" HELYETTESITESE A MINTA HELYERE 206

SYNERR - RATFOR SZINTAXISBELI HIBA JELZESE 300

TABLE - ARCHIVUM TARTALOMJEGYZEKENEK KIIRASA 106

TABPOS - IGAZ, HA COL TABULATORPOZICIO 32

TEXT - SZOVEGSOROK FELDOLGOZASA (ELSO, NYERS VALTOZAT) 239

TEXT - SZOVEGSOROK FELDOLGOZASA (MASODIK IDEIGLENES VALTOZAT) 247

TEXT - SZOVEGSOROK FELDOLGOZASA; VEGLEGES VALTOZAT 254

TPRINT - FILE FEJRESZENEK KIIRASA 106

TRANSLIT - KARAKTEREK LEKEPEZESE 67

TYPE - KARAKTERTIPUS MEGHATAROZASA 142

UNDERL - EGY SOR ALAHUZASA 253

UNIQUE - A SZOMSZEDOS ISMETLODESEK ELHAGYASA 138

UNROT - KWIC ALTAL ROTALT SOROK VISSZAPERMUTALASA 143

UNSTAK - A VEREM URITESE AZ UTASITAS VEGEN 309

UPDATE - MEGLEVO FILE-OK FELUJITASA, UJAK FELVETELE 102

WHILEC - WHILE ELEJEHEZ TARTOZO KOD GENERALASA 316

WHILES - WHILE VEGEHEZ TARTOZO KOD GENERALASA 316

WIDTH - KARAKTERSOROZAT SZELESSEGENEK MEGHATAROZASA 248

WORDCOUNT - SZOSZAMLALAS A STANDARD INPUTON 27

XINDEX - AZ INDEX ALTAL ADOTT ALLAPOT INVERZE 66

XOR - A KIZARO VAGY FUGGVENY 62

Tárgymutató

ábrázolás, adat 120
abs 74
acopy 110
adatábrázolás 120
adatszerkezetek 159, 164, 184, 296, 304, 322
addfil 103
addset 69
Aho, A. V. 173, 326
aktuális sor 177, 188
aláhúzás 51, 229, 231, 252, 253, 255
alapfeltételezések 93, 135, 144, 181, 230, 231, 233, 235
ALGOL 16
algoritmus kiválasztás 37, 118, 131, 134, 166, 173, 256, 272, 292, 304
alldig 301
állítások 125
ALPHA 266, 298
amatch 152, 154, 155, 156
amove 103
and 62
andif utasítás 192, 322
ANSI FORTRAN 42, 322
append 195
ARB 65
archive 99, 195
archívum, file 97
archívumkezelő parancsok 106
arith beépített makro 289
ASCII 25, 265
assert makro 289
átalakítás, jel 257
átírás, karakterek 63
átnevezés, file-ok 103
attribútum, PL/1, rekurzív 155
átütések 51
átvitel, file-ok 103
—, karakterek 19
automatikus nagybetűhasználat 258

BACKSPACE 248
Backus-Naur Forma (BNF) 296
balpar 311
Barron, D. W. 173
beépített makro, arith 289
— —, dnl 271
— —, ifdef 274
— —, ifelse 286
— —, incr 286
— —, index 292
— —, substr 287
— makrók 264, 276, 286
befejeződés, program 115, 223
begin—end 28, 297, 322
beillesztés, file 86, 257, 322, 323
bekezdés, program 26, 29, 41, 194, 297, 323
bekezdések 245
bekezdési mélység 232, 245
belső rendezés 128
beolvasás, karakterek 19
betűtípusokból adódó ellentmondások 99
Beyer, T. 326
bináris keresés 273
blokkdiagramok 112
bonyolultság, időbeli 117, 118, 124, 128, 134, 224, 272
break utasítás 41, 48, 60, 304, 316, 322
brk 249
brknxt 317
Brooks, F. P. 18
Brown, P. J. 293
bubble 117
buborékos rendezés 117
bump makro 277

calc 76
Campbell—Kelly, M. 293
cant 85
carch common block 101

case utasítás 29, 113, 188, 214, 280
catsub 170
cbuf common blokk 197, 222
cdefio common blokk 268
center 253
cfile common blokk 208
change 168, 190
charcount 24, 75
Cherry, L. L. 145, 326
CIA 63
címkék, RATFOR 303
ckglob 212
ckp 201
cline common blokk 185, 300
clook common blokk 272
close 88, 115
clrbuf 196, 198, 223
cmacro common blokk 279
COBOL 16, 150, 323
comand 235
common 139
— blokk, carch 101
— —, cbuf 197, 222
— —, cdefio 268
— —, cfile 208
— —, cline 185, 300
— —, clook 272
— —, cmacro 279
— —, cout 248
— —, coutln 318
— —, cpage 239
— —, cparam 236
— —, cpat 189
— —, cscrat 222
— —, ctxt 189, 198
— utasítás 101, 139, 300
compar 123
compare 81, 84
— makro 286
compiler—compiler 296, 310
compress 56
comtyp 236
concat 89
Constantine, L. L. 18, 115
Cook, J. 326
copen makro 276
copy 20
— (FORTRAN) 22
— (PL/1) 22
Corasick, M. J. 173
count makro 292
cout common blokk 248
coutln common blokk 318
cpage common blokk 239
cparam common blokk 236
cpat common blokk 189
create 96, 114
crypt 55, 61
csatlakozások 75, 77, 86, 139, 140, 147, 171, 234, 323
—, többlépcsős feldolgozás 94
cscrat common blokk 222
csinosítás 323
ctof 76
ctoi 75
CTSS 261, 288
ctxt common blokk 189, 198

Dahl, O. J. 18
Dartmouth Time-sharing System 77
defalt 192
defenzív programozás 57, 143, 157, 160, 193, 240, 328
define 264, 268
deklaráció, integer 20, 24, 28
—, karakter 21, 24, 122, 322
—, logikai 28
—, string 70, 290, 322
deklarációk, komplett 20, 144
delete 110, 202
detab 32
— (FORTRAN) 35
— (PL/1) 37
Deutsch, L. P. 228
difmsg 82
Dijkstra, E. W. 18
DITTO karakter 167, 170, 179
dnl beépített makro 271
do utasítás, FORTRAN 34, 36, 38, 326
— —, PL/1 38
— —, RATFOR 41, 302, 314
—-end 28, 297, 322
— -od 322
docmd 194, 214
docode 314
dodash 72
dodef 284
doglob 213
doif 288
doincr 288

doprnt 193
doread 209
dostat 315
dosub 289
dowrit 210

eatup 315
EBCDIC 25
echo 50
edit 190, 217
egyesítés, file 89
egymásba skatulyázott if 28, 29, 327
elemi műveletek 17, 20, 79, 84, 88, 96, 107, 112, 114, 196, 220, 222, 267, 328
— — összefoglalása, file 114
elemzés 297, 305
elérhetetlen utasítás 305, 322
elhalasztás, kiértékelés 279
elhatároló szóközök 42, 44, 271, 280, 298, 299
elindulás a kályhától 97
elkülönítés, függvények 185, 271
ellenőrzés 289
—, FORTRAN 45
ellentmondások, betűtípusokból adódóak 99
előfeldolgozó, PL/1 makro 86, 293
—, RATFOR 21, 295, 329
előfeldolgozók 16, 86, 323, 326
—, FORTRAN 296, 326
—, makro 322, 326
elrejtés, információ 31, 33, 49, 175, 242, 256, 267, 268, 271, 314, 319, 328
else if utasítás 29, 52, 159, 188, 192, 270, 327
— utasítás 28, 40, 88, 302, 305
elseif 312
elválasztás 259
emberi szemléletmód 14, 82, 85, 95, 109, 117, 131, 175, 204, 230, 252, 258, 260
entab 48
EOF 21, 59, 185, 196, 263
EOS 50
equal 80, 285
ERR 84, 96, 185, 196
error 62, 101
ERROUT 101, 115
és, logikai 52
esc 71
eval 283
exchan 122
expand 55, 60
extrac 109

fcopy 89
fejlesztés, fokozatos 121
feldolgozás (többlépcsős), csatlakozások 94
—, makro 259, 263
—, makrosablonok 264
felhívójel konvenció 71, 149, 258
— — összefoglalás 173
feltételes makrók 260, 274, 286
feltöltött szöveg 230, 247
felülről lefelé haladó programozás 152, 165
— — — tervezés 14, 60, 152, 164, 165, 175, 197
— — — tesztelés 152, 165
filarg 108
file archívum 97
— beillesztés 86, 257, 322, 323
— egyesítés 89
— elemi műveletek összefoglalása 114
— listázó 90
— rendszer 79, 114, 207
file-ok 79
— átnevezése 103
— átvitele 103
— dinamikus létrehozása 96
— másolása 19, 103
— névszerinti egymáshoz rendelése 83
— összehasonlítása 79
—, munka 198, 219, 224
—, random 135
—-rendszer 23
—, PL/1 sysin 23
—, PL/1 sysprint 23
filset 69
find 13, 147, 151, 190
FLECS 326
fokozatos fejlesztés 121
folyamat, összefuttatási 131
folytatási konvenció 42, 295, 314, 317
for utasítás 34, 40, 55, 322
fordítás, kézi 35
fordító, RATFOR—FORTRAN 295
forgatókönyv 175, 195, 226
format 229, 235, 244
formázó összefoglalás 233
—, szöveg 229
FORMFEED 53
FORTRAN do utasítás 34, 36, 38, 326

— ellenőrzés 45
— előfeldolgozók 296, 326
— hibaüzenetek 304, 314, 317
— input/output 26, 81
— karakter összehasonlítás 122
— kulcsszavak 322
— logikai összehasonlítás 28
— portabilitás 295
— vezérlési szerkezetek 38
—, ANSI 42, 322
fprint 91
Fredman, M. L. 115
fsize 105
fskip 107
függvények elkülönítése 185, 271
funkcionális jelölés 264
— leírás 157, 161, 271, 328
— modulok 39, 66, 118, 133, 271
futási idő 37, 72, 134, 166, 256, 273, 275, 291, 304, 324
futó program 140

getarg 49
getc 19, 25, 26, 43, 55, 59, 61, 82, 105, 267, 275
getccl 163
getch 105, 275
getdef 269
getfn 209
getfns 100
gethdr 107
getind 197, 198
getlin 82
getlst 185
getnum 187
getone 186
getpat 161
getrhs 205
getsub 169
gettl 242
gettok 266, 298
gettxt 197, 198, 222
getval 237
getwrd 87
Gimpel, J. F. 173
globális előtag 182, 191, 196, 211, 219, 228
gname 130
gopen 130
goto utasítás 14, 21, 36, 39, 295, 304, 322, 325, 327
gremov 131
Griswold, R. E. 293
gtext 120
gyorsrendezés 124

Hall, A. D. 326
halom 131
hash módszer 271
határesetek 26, 27, 29, 31, 133, 189, 239, 244
hatékonyság 14, 36, 44, 94, 98, 113, 119, 122, 134, 166, 256, 275, 324
head 92
help 100
helyettesítés, szöveg 179, 204
helyettesítési kiválasztás 135
hibaellenőrzés 60, 110, 112, 169, 175, 184, 191, 297, 299
—, RATFOR 309, 323
hibakezelés, RATFOR 322
hibaüzenetek 62, 68, 85, 99, 191, 219, 299
—, FORTRAN 304, 314, 317
hierarchia 112, 165, 190, 200, 217, 218, 245, 254, 320, 328
Hoare, C. A. R. 18, 124, 326
Hopcroft, J. E. 173
HUGE 238
hulladékgyűjtő 204, 224
Hunt, J. W. 115

idő, futási 37, 72, 134, 166, 256, 273, 275, 291, 304, 324
időbeli bonyolultság 117, 118, 124, 128, 134, 224, 272
if utasítás 26, 40, 302, 310
—-else kétértelműség 40, 306, 322
—, egymásba skatulyázott 28, 29, 327
ifcode 311
ifdef beépített makro 274
ifelse beépített makro 286
ifgo 311
IFTRAN 326
illeszkedés 14, 33, 140, 244, 252, 268, 300, 328
illesztő üzemmód 180, 195
imitált program 152, 160, 165, 190, 200, 235
include 88
incr beépített makro 286
index 64
— beépített makro 292
—, kwic 140
—, permutált 140

információ elrejtés 31, 33, 49, 175, 242, 256, 267, 268, 271, 314, 319, 328
init 235
initkw 308
inject 197, 199, 220
input/output, FORTRAN 26, 81
— —, PL/1 26, 81
—, PL/1 stream 26
—, standard 23, 43, 81, 95, 119
instal 273
integer deklaráció 20, 24, 28
írógépszerű szöveg 26, 53
itoc 74

jelátalakítás 257
jelkészlet 25, 62, 69, 77, 122, 143, 322
jelölés, funkcionális 264
Jensen, K. 18
jobbvezérlő nyelv 75, 81, 83, 86, 323
Johnson, S. C. 145, 326
junk változó 70, 130, 162

Kahn, David 77
kályhától való elindulás 97
kapcsolat az operációs rendszerrel 79
kapcsos zárójelek 28, 39, 40, 276, 295, 297, 322
karakter deklaráció 21, 24, 122, 322
— összehasonlítás, FORTRAN 122
— —, PL/1 122
karakterosztály összefoglalás 172
karakter, DITTO 167, 170, 179
karakterek átírása 63
— átvitele 19
— beolvasása 19
karakterosztály 149, 162
karaktersorozat 50
—, üres 50, 150, 161, 169, 260
karakterszámlálás 24
karbantartás 14, 33, 95, 112, 157, 175, 237, 252, 296, 328
katalógus 98
—, központi 98, 114
—, osztott 98, 114
keresés, bináris 273
—, lineáris 131, 133, 272
—, szövegminta 147, 175, 178, 188
Kernighan, B. W. 18, 45, 326
kétértelműség, if-else 40, 306, 322
kettes komplemens 76
keyword-in-context 140
kezdőértékbeállítás 34, 41, 70, 132, 223, 235
kézi fordítás 35
kiegyenlítés, sorvégi 229, 249, 252
kiegyensúlyozott összefuttatás 135
kiértékelés elhalasztása 279
kiértékelési sorrend 57, 60, 188, 192, 265, 270
kifejezések, reguláris 150, 166
kifejtés, makro 263
—, szöveg 59
kiviteli rutinok 318
kizáró—vagy, logikai 62
Knuth, D. E. 135, 145, 172, 293, 326
kódgenerálás 308, 310
kódgenerálási szabályok 301
komplett deklaráció 144
konstansok, szimbolikus 21, 42, 52, 70, 86, 99, 164, 263
konvenció, felhívójel 71, 149, 258
—, folytatási 42, 295, 314, 317
—, megjegyzés 20, 42, 295
korutinok 247
középre állítás 229, 231, 252, 257
központi katalógus 98, 114
kulcsszavak, FORTRAN 322
—, RATFOR 42, 299
külső rendezés 128
kwic 141
— index 140

labelc 313
labgen 313
lábjegyzet 260
Lampson, B. W. 228
leadbl 246
leírás, funkcionális 157, 161, 271, 328
len makro 290
length 66
lépésenkénti finomítás 14, 20, 165
létrehozás, file-ok (dinamikus) 96
lex 300
lexikális analízis 298, 300
lexord 122
lineáris keresés 131, 133, 272
linecount 25, 75
listázó, file 90
locate 159
Loeser, R. 145

logikai deklaráció 28
— egység 23
— és 52
— kifejezések 21, 48, 57, 60, 67, 192
— kizáró—vagy 62
— összehasonlítás, FORTRAN 28
— vagy 28
lookup 272

macro 280
makecopy 96
makfil 130
makhdr 104
maklin 221
makpat 161
makro előfeldolgozó, PL/1 86, 293
— előfeldolgozók 322, 326
— paraméterek 260, 263, 275, 286
—, arith, beépített 289
—, assert 289
—, bump 277
—, compare 286
—, copen 276
—, count 292
—, dnl, beépített 271
—, ifdef, beépített 274
—, ifelse, beépített 286
—, incr, beépített 286
—, index, beépített 292
—, len 290
—, proc 285
—, return 285, 286
—, skipbl 276
—, string 290
—, substr, beépített 287
makrofeldolgozás 259, 263
makrók, beépített 264, 276, 286
—, feltételes 260, 274, 286
makrokifejtés 263
makrosablonok feldolgozása 264
makset 69
maksub 170
másolás, file-ok 19, 103
másolószerkesztő 225
match 152
max 52
McCracken, D. D. 45
McIlroy, M. D. 77, 115, 261
megfejtés 61
megjegyzés konvenció 20, 42, 295
megszakítás 224, 230, 231, 233
mellékhatás 267, 270
méret, program 193
merge 132
metakarakterek 149, 152, 166, 219
Miller, E. 326
min 129
minta lezártja 149, 153, 163
minták keresése, szövegben 147
mod 33
modulkapcsolat-ábra 112
modulok, funkcionális 39, 66, 118, 133, 271
Morris, R. 145
MORTRAN 326
move 203
működési statisztika, RATFOR 325
munka file-ok 198, 219, 224
mutatók 120, 159, 196, 219, 224, 322
Myers, G. J. 18, 115

nagy file-ok rendezése 128
nagy főelem 127
nagybetűhasználat, automatikus 258
névszerinti egymáshoz rendelés, file-ok 83
NEWLINE 25, 26, 43
next utasítás 41, 297, 304, 316, 322
nextln 190
ngetc 267
NO 28
not 62
notfnd 108
NSA 63
nulladik sor 189, 193, 201
nyelvi bővítések 112, 263, 284
nyelvtan, RATFOR 296

OK 185, 196
olvashatóság 37, 39, 328
omatch 157
open 84, 88, 114
operációs rendszerrel való kapcsolat 79
operátorok, reláció 26, 42, 57, 321
optimalizálás 196, 304, 322
—, túl korán 275
optpat 188
or 62
orif utasítás 270, 322
Ossanna, J. F. 261
összefoglalás, felhívójel konvenció 173
—, file elemi műveletek 114

—, formázó 233
—, karakter osztály 172
—, szerkesztő 226
—, szöveg helyettesítés 173
—, szövegminta 172
összefuttatás, kiegyensúlyozott 135
összefutási folyamat 131
összehasonlítás, file-ok 79
összetett utasítás 28
osztott katalógus 98, 114
otherc 317
outch 318
outcon 320
outdon 318
outgo 320
outnum 320
output, standard 23, 43, 81, 95, 109, 119
—, többhasábos 92, 260
outstr 319
outtab 319
overstrike 52

paraméterek, makro 260, 263, 275, 286
paraméterezés, programok 49
parancsok, archívumkezelő 106
Parnas, D. L. 145
parse 307
PASCAL 16, 18
patsiz 157
pbnum 288
pbstr 268
permutált index 140
pfoot 241
phead 240
PL/1 16
— do utasítás 38
— input/output 26, 81
— karakter összehasonlítás 122
— makro előfeldolgozó 86, 293
— résztömbök 284
— stream input 26
— sysin file 23
— sysprint file 23
— vezérlési szerkezetek 38
—, rekurzív attribútum 155
Plauger, P. J. 18
Poage, J. F. 293
Polonsky, I. P. 293
pontosvessző 55, 295
portabilitás 16, 42, 58, 70, 99, 122, 293
—, FORTRAN 295
prevln 190
print 90, 94, 239
proc makro 285
program befejeződés 115, 223
— bekezdés 26, 29, 41, 194, 297, 323
— méret 193
— tesztelés 29
—, futó 140
—, imitált 152, 160, 165, 190, 200, 235
—, ravasz 251
—, strukturált 165, 185, 190, 196, 235
programok paraméterezése 49
programozás, felülről lefelé haladó 152, 165
programszerkezet 112
programozás, defenzív 57, 143, 157, 160, 193, 240, 328
—, strukturált 14, 39, 328
—, személytelen 14
pszeudokód 20
ptext 121
ptscan 189
puffer-reprezentáció 195
pufferkezelés 197
push 283
put 240
putbak 267
putbuf 56
putc 19, 25, 26, 43, 55, 59, 61, 82, 264, 275
putch 110, 264, 275
putchr 282
putdec 24, 73
putlin 83
putrot 141
puttl 241
puttok 282
putwrd 247, 250

quick 126

random file-ok 135
RATFOR 309
— címkék 303
— do utasítás 41, 302, 314
— előfeldolgozó 21, 295, 329
— hibaellenőrzés 309, 323
— hibakezelés 322
— kulcsszavak 299
— működési statisztika 325
— nyelvtan 296

— összefoglalás 39
— utasítások 42
— vezérlési szerkezetek 39, 301
— —FORTRAN fordító 295
ravasz program 251
READ 84
readf 222, 223
READWRITE 84
reguláris kifejezések 150, 166
reheap 133
rejtjelezés 61
rekurzió 173, 266, 268, 277, 297
rekurzív attribútum, PL/1 155
rekurzivitás 124, 154, 164, 219
reláció operátorok 26, 42, 57, 321
relink 197, 199
remark 82, 101
remove 95, 103, 115
rendezés 117
—, belső 128
—, buborékos 117
—, gyors 124
—, külső 128
—, nagy file-ok 128
—, shell 118
—, szövegek 119
rendszám 128
rendszer, file 23, 79, 114, 207
repeat utasítás 33, 41, 48, 322
replac 111
résztömbök, PL/1 284
return makro 285, 286
— utasítás 304, 322
Rice, D. E. 288
Rich, R. P. 145
Ritchie, D. M. 77, 115, 288, 293, 326
ROFF 261
rotate 142
run 224
RUNOFF 261
rutinok, kiviteli 318
Ryder, B. G. 45

Saltzer, J. 261
scopy 104
Sedgewick, R. 145
seek 220
set 238
setbuf 196, 199, 223
settab 33
— (FORTRAN) 35
— (PL/1) 38
shell 118, 121
— rendezés 118
skip 26, 92
skipbl 187
— makro 276
SNOBOL 173, 293, 323
sor, aktuális 177, 188
—, utolsó 177
sornyomtató vezérlő karakter 51
sorozat, szöveghelyettesítő 167
sorrend, kiértékelési 57, 60, 188, 192, 265, 270
sorszámlálás 25, 231
sort 120, 129
sorvégi kiegyenlítés 229, 249, 252
space 243
spread 250
standard input 23, 43, 81, 95, 119
— output 23, 43, 81, 95, 109, 119
stclos 163
STDIN 43, 83, 101, 105, 115
STDOUT 43, 83, 101, 115
Stevens, W. P. 18, 115
stop utasítás 44, 304, 322
stream input, PL/1 26
string deklaráció 70, 290, 322
— makro 290
struktúráló 323
strukturált program 165, 185, 190, 196, 235
— programozás 14, 39, 328
— tervezés 14
subst 206
substr beépített makro 287
svájci katonai kés 93
synerr 300
sysin file, PL/1 23
sysprint file, PL/1 23
számjel 241
számlálás, karakterek 24
—, sorok 25, 231
—, szavak 27
számok 73
szavak számlálása 27
szedés (nyomdai) 229, 261, 326
szemantikai tevékenység 297
személytelen programozás 14
szerkesztés 175
szerkesztő, másoló 225

—, összefoglalás 226
—, szöveg 147, 175
szerkezet, program 112
—, vezérlési 112
szerkezetek, adat 159, 164, 184, 296, 304, 322
—, FORTRAN vezérlési 38
—, PL/1 vezérlési 38
—, RATFOR vezérlési 39, 301
—, vezérlési 14, 26, 28, 33, 36, 295
szimbolikus konstansok 21, 42, 52, 70, 86, 99, 164, 263
szimbólum 264, 266, 298
szóközök, elhatároló 42, 44, 271, 280, 298, 299
szolgáltatások 92, 95, 99, 161, 260
szöveg feltöltött 230, 247
szövegformázó 229
szöveg helyettesítés 179, 204
— — összefoglalás 173
szövegszerkesztő 147, 175
— áttekintés 175
szöveg-egység 140
—, írógépszerű 26, 53
szövegek rendezése 119
szöveghelyettesítő sorozat 167
szövegkifejtés 59
szövegminta keresés 147, 175, 178, 188
—, összefoglalás 172
szövegtömörítő 53
szövegváltoztatás 167
szűrők 16, 47, 77, 119

TAB 28
table 106
tabpos 32
— (FORTRAN) 36
— (PL/1) 38
tabulátorok eltávolítása 30
— visszaállítása 47
tail 76
TECO 228
telefon teszt 67
tervezés, felülről lefelé haladó 14, 60, 152, 164, 165, 175, 197
tervezés, strukturált 14
tesztelés, felülről lefelé haladó 152, 165
—, program 29
text 239, 247, 254
Thompson, K. L. 77, 115, 173, 228
többhasábos output 92, 260
többlépcsős feldolgozás, csatlakozások 94
többutas döntés 29, 184, 235, 280
— elágazás 99, 71
tömörítő, szöveg 53
tprint 106
translit 67, 149, 190, 207
TROFF 261
túl korai optimalizálás 275
type 142

újsor karakter 199
Ullman, J. D. 173, 326
underl 253
unique 138
UNIX 77, 115, 173, 175, 261, 326
unrot 143
unstak 309
until utasítás 33, 41, 322
update 102
üres karaktersorozat 50, 150, 161, 169, 260
— utasítás 55
utasítás, andif 192, 322
—, break 41, 48, 60, 304, 316, 322
—, case 29, 113, 188, 214, 280
—, common 101, 139, 300
—, elérhetetlen 305, 322
—, else 28, 40, 88, 302, 305
—, — if 29, 52, 57, 159, 188, 192, 270, 327
—, for 34, 40, 55, 322
—, FORTRAN do 34, 36, 38, 326
—, goto 14, 21, 36, 39, 295, 304, 322, 325, 327
—, if 26, 40, 302, 310
—, next 41, 297, 304, 316, 322
—, orif 270, 322
—, összetett 28
—, PL/1 do 38
—, RATFOR do 41, 302, 314
—, repeat 33, 41, 48, 322
—, return 304, 322
—, stop 44, 304, 322
—, until 33, 41, 322
—, üres 55
—, while 21, 22, 25, 40, 303, 316
utasítások, vezérlő 21
utolsó sor 177
üzenetek 82, 109, 193, 194, 209, 218
—, hiba 62, 68, 85, 99, 191, 219

vagy, logikai 28

változó, junk 70, 130, 162
változtatás, szöveg 167
Van Dam, A. 288
vezérlési szerkezetek 14, 22, 26, 28, 33, 36, 295
— —, FORTRAN 38
— —, PL/1 38
— —, RATFOR 39, 301
vezérlő utasítások 21
visszaállítás, tabulátorok 47
visszalépés 51
—-karakter (BACKSPACE) 248
Weinberg, G. M. 18
while utasítás 21, 22, 25, 40, 303, 316
whilec 316
whiles 316
width 248
Wirth, N. 18
wordcount 27, 76
WRITE 84

xindex 66
xor 62

YES 28
Yourdon, E. 18

Zahn, C. 326

KÖSSE ÖSSZE KISSZÁMÍTÓGÉPÉT A SZÜV GÉPEIVEL, ÚJ LEHETŐSÉGEKHEZ JUT:

- PL/1 Optimizer
- PICS, PMS IV., MARK IV.
- MPS, ICES, SSP, GPSS
 (automatizált mérnöki tervezés)
- Osztott feldolgozás

Host gépek: R–22, R–35
Operációs rendszer: IBM OS HASP
Hostok helye: Budapest
Debrecen
Győr
Kaposvár
Pécs
Székesfehérvár

BÍZZA MEG A SZÜV-ÖT A RENDSZER KIÉPÍTÉSÉVEL

SZÜV

Felvilágosítás: KSH SZÜV Alkalmazásfejlesztés
1145 Budapest XIV., Szugló u. 9–15.
Telefon: 634–095
Telex: 22–6216

Kiadja a Műszaki Könyvkiadó
Felelős Kiadó: Fischer Herbert igazgató
Szedte az Alföldi Nyomda
Nyomta az Alföldi Nyomda
A nyomdai megrendelés törzsszáma: 1500.66-13-1
Készült Debrecenben, 1982-ben
Műszaki vezető: Kőrizs Károly
Műszaki szerkesztő: Mózer István
A borítót és a kötést tervezte: Sebes János
A könyv formátuma: B/5 — Ívterjedelme: 31,2 (A/5)
Példányszám: 2450
Papír minősége: 80 g ofszet — Betűcsalád és -méret: New Times gm/gm
Azonossági szám: 61 034 — **Mű:** 3257–k–8284
Készült az MSZ 5601 és 5602 szerint
Kézirat lezárva: 1981. január

350.-